Siddhartha Rao

EIGHTH EDITION

New C++14 & C++17 Coverage

Sams Teach Yourself C++ in One Hour a Day

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Siddhartha Rao

Sams**TeachYourself**

C++

in **One Hour** a Day

Eighth Edition

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

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

ISBN-13: 978-0-7897-5774-6

ISBN-10: 0-7897-5774-5

Library of Congress Control Number: 2016958138

First Printing: December 2016

Trademarks

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

Warning and Disclaimer

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

Special Sales

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

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Editor
Mark Taber

Senior Project Editor
Tonya Simpson

Copy Editor
Geneil Breeze

Indexer
Erika Millen

Proofreader
Sasirekha Durairajan

Technical Editor
Adrian Ngo

Compositor
codeMantra

Contents

Dedication

In memory of my father, who will continue to be my source of inspiration.

Acknowledgments

I am thankful to my family for the immense support, to my wife Clara, and to the editorial staff for their spirited engagement in getting this book to you!

About the Author

Siddhartha Rao is the Vice President in charge of Security Response at SAP SE, the world's leading supplier of enterprise software. The evolution of C++ convinces Siddhartha that you can program faster, simpler, and more powerful applications than ever before. He loves traveling and is a passionate mountain biker. He looks forward to your feedback on this effort!

We Want to Hear from You!

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

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

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

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

Email: feedback@samspublishing.com

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

Reader Services

Visit the publisher's website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that may be available for this book.

Introduction

2011 and 2014 were two special years for C++. While C++11 ushered in a dramatic improvement to C++, introducing new keywords and constructs that increased your programming efficiency, C++14 brought in incremental improvements that added finishing touches to the features introduced by C++11.

This book helps you learn C++ in tiny steps. It has been thoughtfully divided into lessons that teach you the fundamentals of this object-oriented programming language from a practical point of view. Depending on your proficiency level, you will be able to master C++ one hour at a time.

Learning C++ by doing is the best way—so try the rich variety of code samples in this book hands-on and help yourself improve your programming proficiency. These code snippets have been tested using the latest versions of the available compilers at the time of writing, namely the Microsoft Visual C++ compiler for C++ and GNU's C++ compiler, which both offer a rich coverage of C++14 features.

Who Should Read This Book?

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

NOTE

Visit the publisher's website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that may be available for this book.

Organization of This Book

Depending on your current proficiency levels with C++, you can choose the section you would like to start with. Concepts introduced by C++11 and C++14 are sprinkled throughout the book, in the relevant lessons. This book has been organized into five parts:

- Part I, "The Basics," gets you started with writing simple C++ applications. In doing so, it introduces you to the keywords that you most frequently see in C++ code of a variable without compromising on type safety.
- Part II, "Fundamentals of Object-Oriented C++ Programming," teaches you the concept of classes. You learn how C++ supports the important object-oriented programming principles of encapsulation, abstraction, inheritance, and polymorphism. Lesson 9, "Classes and Objects," teaches you the concept of move constructor followed by the move assignment operator in Lesson 12, "Operator Types and Operator Overloading." These performance features help reduce unwanted and unnecessary copy steps, boosting the performance of your application. Lesson 14, "An Introduction to Macros and Templates," is your stepping stone to writing powerful generic C++ code.
- Part III, "Learning the Standard Template Library (STL)," helps you write efficient and practical C++ code using the STL `string` class and containers. You learn how `std::string` makes simple string concatenation operations safe and easy and how you don't need to use C-style `char*` strings anymore. You will be able to use STL dynamic arrays and linked lists instead of programming your own.
- Part IV, "More STL," focuses on algorithms. You learn to use `sort` on containers such as `vector` via iterators. In this part, you find out how keyword `auto` introduced by C++11 has made a significant reduction to the length of your iterator declarations. Lesson 22, "Lambda Expressions," presents a powerful new feature that results in significant code reduction when you use STL algorithms.
- Part V, "Advanced C++ Concepts," explains language capabilities such as smart pointers and exception handling, which are not a must in a C++ application but help make a significant contribution toward increasing its stability and quality. This part ends with a note on best practices in writing good C++ applications, and introduces you to the new features expected to make it to the next version of the ISO standard called C++17.

Conventions Used in This Book

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

NOTE

These boxes provide additional information related to material you read.

CAUTION

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

TIP

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

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

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

Sample Code for This Book

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

This page intentionally left blank

LESSON 3

Using Variables, Declaring Constants

Variables are tools that help the programmer temporarily store data for a finite amount of time. *Constants* are tools that help the programmer define artifacts that are not allowed to change or make changes.

In this lesson, you find out

- How to declare and define variables and constants
- How to assign values to variables and manipulate those values
- How to write the value of a variable to the screen
- How to use keywords `auto` and `constexpr`

What Is a Variable?

Before you actually explore the need and use of variables in a programming language, take a step back and first see what a computer contains and how it works.

Memory and Addressing in Brief

All computers, smart phones, and other programmable devices contain a microprocessor and a certain amount of memory for temporary storage called Random Access Memory (RAM). In addition, many devices also allow for data to be persisted on a storage device such as the hard disk. The microprocessor executes your application, and in doing so it works with the RAM to fetch the application binary code to be executed as well as the data associated with it, which includes that displayed on the screen and that entered by the user.

The RAM itself can be considered to be a storage area akin to a row of lockers in the dorms, each locker having a number—that is, an address. To access a location in memory, say location 578, the processor needs to be asked via an instruction to fetch a value from there or write a value to it.

Declaring Variables to Access and Use Memory

The following examples will help you understand what variables are. Assume you are writing a program to multiply two numbers supplied by the user. The user is asked to feed the multiplicand and the multiplier into your program, one after the other, and you need to store each of them so that you can use them later to multiply. Depending on what you want to be doing with the result of the multiplication, you might even want to store it for later use in your program. It would be slow and error-prone if you were to explicitly specify memory addresses (such as 578) to store the numbers, as you would need to worry about inadvertently overwriting existing data at the location or your data being overwritten at a later stage.

When programming in languages like C++, you define variables to store those values. Defining a variable is quite simple and follows this pattern:

```
VariableType VariableName;
```

or

```
VariableType VariableName = InitialValue;
```

The variable type attribute tells the compiler the nature of data the variable can store, and the compiler reserves the necessary space for it. The name chosen by the programmer is a friendly replacement for the address in the memory where the variable's value is stored.

Unless the initial value is assigned, you cannot be sure of the contents of that memory location, which can be bad for the program. Therefore, initialization is optional, but it's often a good programming practice. Listing 3.1 shows how variables are declared, initialized, and used in a program that multiplies two numbers supplied by the user.

LISTING 3.1 Using Variables to Store Numbers and the Result of Their Multiplication

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main ()
 5: {
 6:     cout << "This program will help you multiply two numbers" << endl;
 7:
 8:     cout << "Enter the first number: ";
 9:     int firstNumber = 0;
10:     cin >> firstNumber;
11:
12:     cout << "Enter the second number: ";
13:     int secondNumber = 0;
14:     cin >> secondNumber;
15:
16:     // Multiply two numbers, store result in a variable
17:     int multiplicationResult = firstNumber * secondNumber;
18:
19:     // Display result
20:     cout << firstNumber << " x " << secondNumber;
21:     cout << " = " << multiplicationResult << endl;
22:
23:     return 0;
24: }
```

Output ▼

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

Analysis ▼

This application asks the user to enter two numbers, which the program multiplies and displays the result. To use numbers entered by the user, it needs to store them in the memory. Variables `firstNumber` and `secondNumber` declared in Lines 9 and 13 do the job of temporarily storing integer values entered by the user. You use `std::cin` in Lines 10 and 14 to accept input from the user and to store them in the two integer variables. The `cout` statement in Line 21 is used to display the result on the console.

Analyzing a variable declaration further:

```
9:     int firstNumber = 0;
```

What this line declares is a variable of type `int`, which indicates an integer, with a name called `firstNumber`. Zero is assigned to the variable as an initial value.

The compiler does the job of mapping this variable `firstNumber` to a location in memory and takes care of the associated memory-address bookkeeping for you for all the variables that you declare. The programmer thus works with human-friendly names, while the compiler manages memory-addressing and creates the instructions for the microprocessor to execute in working with the RAM.

CAUTION

Naming variables appropriately is important for writing good, understandable, and maintainable code.

Variable names in C++ can be alphanumeric, but they cannot start with a number. They cannot contain spaces and cannot contain arithmetic operators (such as `+`, `–`, and so on) within them. Variable names also cannot be reserved keywords. For example, a variable named `return` will cause compilation failure.

Variable names can contain the underscore character_that often is used in descriptive variable naming.

Declaring and Initializing Multiple Variables of a Type

In Listing 3.1, `firstNumber`, `secondNumber`, and `multiplicationResult` are all of the same type—integers—and are declared in three separate lines. If you wanted to, you could condense the declaration of these three variables to one line of code that looks like this:

```
int firstNumber = 0, secondNumber = 0, multiplicationResult = 0;
```

NOTE

As you can see, C++ makes it possible to declare multiple variables of a type at once and to declare variables at the beginning of a function. Yet, declaring a variable when it is first needed is often better as it makes the code readable—one notices the type of the variable when the declaration is close to its point of first use.

CAUTION

Data stored in variables is data stored in RAM. This data is lost when the application terminates unless the programmer explicitly persists the data on a storage medium like a hard disk.

Storing to a file on disk is discussed in Lesson 27, "Using Streams for Input and Output."

Understanding the Scope of a Variable

Ordinary variables like the ones we have declared this far have a well-defined scope within which they're valid and can be used. When used outside their scope, the variable names will not be recognized by the compiler and your program won't compile. Beyond its scope, a variable is an unidentified entity that the compiler knows nothing of.

To better understand the scope of a variable, reorganize the program in Listing 3.1 into a function `MultiplyNumbers()` that multiplies the two numbers and returns the result. See Listing 3.2.

LISTING 3.2 Demonstrating the Scope of the Variables

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: void MultiplyNumbers ()
 5: {
 6:     cout << "Enter the first number: ";
 7:     int firstNumber = 0;
 8:     cin >> firstNumber;
 9:
10:     cout << "Enter the second number: ";
11:     int secondNumber = 0;
12:     cin >> secondNumber;
13:
14:     // Multiply two numbers, store result in a variable
15:     int multiplicationResult = firstNumber * secondNumber;
16:
17:     // Display result
18:     cout << firstNumber << " x " << secondNumber;
19:     cout << " = " << multiplicationResult << endl;
20: }
21: int main ()
22: {
23:     cout << "This program will help you multiply two numbers" << endl;
24:
25:     // Call the function that does all the work
26:     MultiplyNumbers();
```

```
27:
28:     // cout << firstNumber << " x " << secondNumber;
29:     // cout << " = " << multiplicationResult << endl;
30:
31:     return 0;
32: }
```

Output ▼

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

Analysis ▼

Listing 3.2 does exactly the same activity as Listing 3.1 and produces the same output. The only difference is that the bulk of the work is delegated to a function called `MultiplyNumbers()` invoked by `main()`. Note that variables `firstNumber` and `secondNumber` cannot be used outside of `MultiplyNumbers()`. If you uncomment Lines 28 or 29 in `main()`, you experience compile failure of type `undeclared identifier`.

This is because the scope of the variables `firstNumber` and `secondNumber` is local, hence limited to the function they're declared in, in this case `MultiplyNumbers()`. A local variable can be used in a function after variable declaration till the end of the function. The curly brace (}) that indicates the end of a function also limits the scope of variables declared in the same. When a function ends, all local variables are destroyed and the memory they occupied returned.

When compiled, variables declared within `MultiplyNumbers()` perish when the function ends, and if they're used in `main()`, compilation fails as the variables have not been declared in there.

CAUTION

If you declare another set of variables with the same name in `main()`, then don't still expect them to carry a value that might have been assigned in `MultiplyNumbers()`.

The compiler treats the variables in `main()` as independent entities even if they share their names with a variable declared in another function, as the two variables in question are limited by their scope.

Global Variables

If the variables used in function `MultiplyNumbers()` in Listing 3.2 were declared outside the scope of the function `MultiplyNumber()` instead of within it, then they would be usable in both `main()` and `MultiplyNumbers()`. Listing 3.3 demonstrates global variables, which are the variables with the widest scope in a program.

LISTING 3.3 Using Global Variables

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // three global integers
 5: int firstNumber = 0;
 6: int secondNumber = 0;
 7: int multiplicationResult = 0;
 8:
 9: void MultiplyNumbers ()
10: {
11:     cout << "Enter the first number: ";
12:     cin >> firstNumber;
13:
14:     cout << "Enter the second number: ";
15:     cin >> secondNumber;
16:
17:     // Multiply two numbers, store result in a variable
18:     multiplicationResult = firstNumber * secondNumber;
19:
20:     // Display result
21:     cout << "Displaying from MultiplyNumbers(): ";
22:     cout << firstNumber << " x " << secondNumber;
23:     cout << " = " << multiplicationResult << endl;
24: }
25: int main ()
26: {
27:     cout << "This program will help you multiply two numbers" << endl;
28:
29:     // Call the function that does all the work
30:     MultiplyNumbers();
31:
32:     cout << "Displaying from main(): ";
33:
34:     // This line will now compile and work!
35:     cout << firstNumber << " x " << secondNumber;
36:     cout << " = " << multiplicationResult << endl;
37:
38:     return 0;
39: }
```

Output ▼

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 19
Displaying from MultiplyNumbers(): 51 x 19 = 969
Displaying from main(): 51 x 19 = 969
```

Analysis ▼

Listing 3.3 displays the result of multiplication in two functions, neither of which has declared the variables `firstNumber`, `secondNumber`, and `multiplicationResult`. These variables are `global` as they have been declared in Lines 5–7, outside the scope of any function. Note Lines 23 and 36 that use these variables and display their values. Pay special attention to how `multiplicationResult` is first assigned in `MultiplyNumbers()` yet is effectively reused in `main()`.

CAUTION

Indiscriminate use of global variables is considered poor programming practice. This is because global variables can be assigned values in any/every function and can contain an unpredictable state, especially when functions that modify them run in different threads or are programmed by different programmers in a team.

An elegant way of programming Listing 3.3 without using global variables would have the function `MultiplyNumbers()` return the integer result of the multiplication to `main()`.

Naming Conventions

In case you haven't noticed, we named the function `MultiplyNumbers()` where every word in the function name starts with a capital letter (called *Pascal casing*), while variables `firstNumber`, `secondNumber`, and `multiplicationResult` were given names where the first word starts with a lowercase letter (called *camel casing*). This book follows a convention where variable names follow camel casing, while other artifacts such as function names follow Pascal casing.

You may come across C++ code wherein a variable name is prefixed with characters that explain the type of the variable. This convention is called the *Hungarian notation*

and is frequently used in the programming of Windows applications. So, `firstNumber` in Hungarian notation would be `iFirstNumber`, where the prefix `i` stands for integer. A global integer would be called `g_iFirstNumber`. Hungarian notation has lost popularity in recent years in part due to improvements in Integrated Development Environments (IDEs) that display the type of a variable when required—on mouse hover, for instance.

Examples of commonly found bad variable names follow:

```
int i = 0;
bool b = false;
```

The name of the variable should indicate its purpose, and the two can be better declared as

```
int totalCash = 0;
bool isLampOn = false;
```


CAUTION

Naming conventions are used to make the code readable to programmers, not to compilers. So choose a convention that suits wisely and use it consistently.

When working in a team, it is a good idea to align on the convention to be used before starting a new project. When working on an existing project, adopt the used convention so that the new code remains readable to others.

Common Compiler-Supported C++ Variable Types

In most of the examples thus far, you have defined variables of type `int`—that is, integers. However, C++ programmers can choose from a variety of fundamental variable types supported directly by the compiler. Choosing the right variable type is as important as choosing the right tools for the job! A Phillips screwdriver won't work well with a regular screw head just like an unsigned integer can't be used to store values that are negative! Table 3.1 enlists the various variable types and the nature of data they can contain.

TABLE 3.1 Variable Types

Type	Values
`bool`	`true` or `false`
`char`	256 character values
`unsigned short int`	0 to 65,535
`short int`	–32,768 to 32,767
`unsigned long int`	0 to 4,294,967,295
`long int`	–2,147,483,648 to 2,147,483,647
`unsigned long long`	0 to 18,446,744,073,709,551,615
`long long`	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`int` (16 bit)	–32,768 to 32,767
`int` (32 bit)	–2,147,483,648 to 2,147,483,647
`unsigned int` (16 bit)	0 to 65,535
`unsigned int` (32 bit)	0 to 4,294,967,295
`float`	1.2e–38 to 3.4e38
`double`	2.2e–308 to 1.8e308

The following sections explain the important types in greater detail.

Using Type `bool` to Store Boolean Values

C++ provides a type that is specially created for containing Boolean values `true` or `false`, both of which are reserved C++ keywords. This type is particularly useful in storing settings and flags that can be ON or OFF, present or absent, available or unavailable, and the like.

A sample declaration of an initialized Boolean variable is

```
bool alwaysOnTop = false;
```

An expression that evaluates to a Boolean type is

```
bool deleteFile = (userSelection == "yes");
// evaluates to true if userSelection contains "yes", else to false
```

Conditional expressions are explained in Lesson 5, “Working with Expressions, Statements, and Operators.”

Using Type `char` to Store Character Values

Use type `char` to store a single character. A sample declaration is

```
char userInput = 'Y'; // initialized char to 'Y'
```

Note that memory is comprised of bits and bytes. Bits can be either 0 or 1, and bytes can contain numeric representation using these bits. So, working or assigning character data as shown in the example, the compiler converts the character into a numeric representation that can be placed into memory. The numeric representation of Latin characters A–Z, a–z, numbers 0–9, some special keystrokes (for example, DEL), and special characters (such as backspace) has been standardized by the American Standard Code for Information Interchange, also called ASCII.

You can look up the table in Appendix D, "ASCII Codes," to see that the character Y assigned to variable `userInput` has the ASCII value 89 in decimal. Thus, what the compiler does is store 89 in the memory space allocated for `userInput`.

The Concept of Signed and Unsigned Integers

Sign implies positive or negative. All numbers you work with using a computer are stored in the memory in the form of bits and bytes. A memory location that is 1 byte large contains 8 bits. Each bit can either be a 0 or 1 (that is, carry one of these two values at best). Thus, a memory location that is 1 byte large can contain a maximum of 2 to the power 8 values—that is, 256 unique values. Similarly, a memory location that is 16 bits large can contain 2 to the power 16 values—that is, 65,536 unique values.

If these values were to be unsigned—assumed to be only positive—then one byte could contain integer values ranging from 0 through 255 and two bytes would contain values ranging from 0 through 65,535, respectively. Look at Table 3.1 and note that the `unsigned short` is the type that supports this range, as it is contained in 16 bits of memory. Thus, it is quite easy to model positive values in bits and bytes (see Figure 3.1).

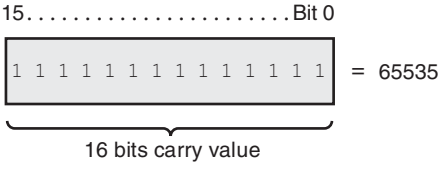

FIGURE 3.1
Organization of bits in a 16-bit unsigned short integer.

How to model negative numbers in this space? One way is to "sacrifice" a bit as the sign-bit that would indicate if the values contained in the other bits are positive or

negative (see Figure 3.2). The sign-bit needs to be the most-significant-bit (MSB) as the least-significant-one would be required to model odd numbers. So, when the MSB contains sign-information, it is assumed that 0 would be positive and 1 would mean negative, and the other bytes contain the absolute value.

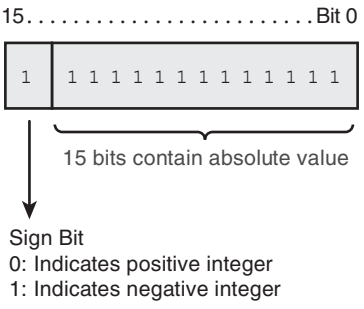

FIGURE 3.2
Organization of bits in a 16-bit signed short integer.

Thus, a signed number that occupies 8 bits can contain values ranging from –128 through 127, and one that occupies 16 bits can contain values ranging from –32,768 through 32,767. If you look at Table 3.1 again, note that the `(signed) short` is the type that supports positive and negative integer values in a 16-bit space.

Signed Integer Types `short`, `int`, `long`, and `long long`

These types differ in their sizes and thereby differ in the range of values they can contain. `int` is possibly the most used type and is 32 bits wide on most compilers. Use the right type depending on your projection of the maximum value that particular variable would be expected to hold.

Declaring a variable of a signed type is simple:

```
short int gradesInMath = -5; // not your best score
int moneyInBank = -70000; // overdraft
long populationChange = -85000; // reducing population
long long countryGDPChange = -70000000000;
```

Unsigned Integer Types `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`

Unlike their signed counterparts, unsigned integer variable types cannot contain sign information, and hence they can actually support twice as many positive values.

Declaring a variable of an unsigned type is as simple as this:

```
unsigned short int numColorsInRainbow = 7;
unsigned int numEggsInBasket = 24; // will always be positive
unsigned long  numCarsInNewYork = 700000;
unsigned long long countryMedicareExpense = 70000000000;
```

NOTE

You would use an unsigned variable type when you expect only positive values. So, if you're counting the number of apples, don't use `int`; use `unsigned int`. The latter can hold twice as many values in the positive range as the former can.

CAUTION

So, an unsigned type might not be suited for a variable in a banking application used to store the account balance as banks do allow some customers an overdraft facility. To see an example that demonstrates the differences between signed and unsigned types, visit Listing 5.3 in Lesson 5.

Avoid Overflow Errors by Selecting Correct Data Types

Data types such as `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, and the like have a finite capacity for containing numbers. When you exceed the limit imposed by the type chosen in an arithmetic operation, you create an overflow.

Take `unsigned short` for an example. Data type `short` consumes 16 bits and can hence contain values from 0 through 65,535. When you add 1 to 65,535 in an `unsigned short`, the value overflows to 0. It's like the odometer of a car that suffers a mechanical overflow when it can support only five digits and the car has done 99,999 kilometers (or miles).

In this case, `unsigned short` was never the right type for such a counter. The programmer was better off using `unsigned int` to support numbers higher than 65,535.

In the case of a `signed short` integer, which has a range of –32,768 through 32,767, adding 1 to 32,767 may result in the `signed integer` taking the highest negative value. This behavior is compiler dependent.

Listing 3.4 demonstrates the overflow errors that you can inadvertently introduce via arithmetic operations.

LISTING 3.4 Demonstrating the Ill-Effects of Signed and Unsigned Integer Overflow Errors

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     unsigned short uShortValue = 65535;
 6:     cout << "Incrementing unsigned short " << uShortValue << " gives: ";
 7:     cout << ++uShortValue << endl;
 8:
 9:     short signedShort = 32767;
10:     cout << "Incrementing signed short " << signedShort << " gives: ";
11:     cout << ++signedShort << endl;
12:
13:     return 0;
14: }
```

Output ▼

```
Incrementing unsigned  short 65535 gives: 0
Incrementing signed short 32767 gives: -32768
```

Analysis ▼

The output indicates that unintentional overflow situations result in unpredictable and unintuitive behavior for the application. Lines 7 and 11 increment an `unsigned short` and a `signed short` that have previously been initialized to their maximum supported values –65,535 and 32,767, respectively. The output demonstrates the value they hold after the increment operation, namely an overflow of 65,535 to zero in the unsigned short and an overflow of 32,767 to –32,768 in the signed short. One wouldn't expect the result of an increment operation to reduce the value in question, but that is exactly what happens when an integer type overflows. If you were using the values in question to allocate memory, then with the `unsigned short`, you can reach a point where you request zero bytes when your actual need is 65536 bytes.

NOTE

The operations `++uShortValue` and `++signedShort` seen in Listing 3.4 at lines 7 and 11 are prefix increment operations. These are explained in detail in Lesson 5.

Floating-Point Types `float` and `double`

Floating-point numbers are what you might have learned in school as real numbers. These are numbers that can be positive or negative. They can contain decimal values. So, if you want to store the value of pi (22 / 7 or 3.14) in a variable in C++, you would use a floating-point type.

Declaring variables of these types follows exactly the same pattern as the `int` in Listing 3.1. So, a `float` that allows you to store decimal values would be declared as the following:

```
float pi = 3.14;
```


And a double precision `float` (called simply a double) is defined as

```
double morePrecisePi = 22.0 / 7;
```

TIP

C++14 adds support for *chunking separators* in the form of a single quotation mark. This improves readability of code, as seen in the following initializations:

```
int moneyInBank = -70'000; // -70000
long populationChange = -85'000; // -85000
long long countryGDPChange = -70'000'000'000; //
-70 billion
double pi = 3.141'592'653'59; // 3.14159265359
```

NOTE

The data types mentioned thus far are often referred to as POD (Plain Old Data). The category POD contains these as well as aggregations (structs, enums, unions, or classes) thereof.

Determining the Size of a Variable Using `sizeof`

Size is the amount of memory that the compiler reserves when the programmer declares a variable to hold the data assigned to it. The size of a variable depends on its type, and C++ has a very convenient operator called `sizeof` that tells you the size in bytes of a variable or a type.

The usage of `sizeof` is simple. To determine the size of an integer, you invoke `sizeof` with parameter `int` (the type) as demonstrated by Listing 3.5.

```
cout << "Size of an int: " << sizeof (int);
```

LISTING 3.5 Finding the Size of Standard C++ Variable Types

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     using namespace std;
 6:     cout << "Computing the size of some C++ inbuilt variable types" << endl;
 7:
 8:     cout << "Size of bool: " << sizeof(bool) << endl;
 9:     cout << "Size of char: " << sizeof(char) << endl;
10:     cout << "Size of unsigned short int: " << sizeof(unsigned short) << endl;
11:     cout << "Size of short int: " << sizeof(short) << endl;
12:     cout << "Size of unsigned long int: " << sizeof(unsigned long) << endl;
13:     cout << "Size of long: " << sizeof(long) << endl;
14:     cout << "Size of int: " << sizeof(int) << endl;
15:     cout << "Size of unsigned long long: "<< sizeof(unsigned long long)<<
endl;
16:     cout << "Size of long long: " << sizeof(long long) << endl;
17:     cout << "Size of unsigned int: " << sizeof(unsigned int) << endl;
18:     cout << "Size of float: " << sizeof(float) << endl;
19:     cout << "Size of double: " << sizeof(double) << endl;
20:
21:     cout << "The output changes with compiler, hardware and OS" << endl;
22:
23:     return 0;
24: }
```

Output ▼

```
Computing the size of some C++ inbuilt variable types
Size of bool: 1
Size of char: 1
Size of unsigned short int: 2
Size of short int: 2
Size of unsigned long int: 4
Size of long: 4
Size of int: 4
Size of unsigned long long: 8
Size of long long: 8
Size of unsigned int: 4
Size of float: 4
Size of double: 8
The output changes with compiler, hardware and OS
```

Analysis ▼

The output of Listing 3.5 reveals sizes of various types in bytes and is specific to my platform: compiler, OS, and hardware. This output in particular is a result of running the program in 32-bit mode (compiled by a 32-bit compiler) on a 64-bit operating system. Note that a 64-bit compiler probably creates different results, and the reason I chose a 32-bit compiler was to be able to run the application on 32-bit as well as 64-bit systems. The output tells that the `sizeof` a variable doesn't change between an unsigned or signed type; the only difference in the two is the MSB that carries sign information in the former.

NOTE

All sizes seen in the output are in bytes. The size of a type is an important parameter to be considered, especially for types used to hold numbers. A `short int` can hold a smaller range than a `long long`. You therefore wouldn't use a `short int` to hold the population of a country, for example.

TIP

C++11 introduced fixed-width integer types that allow you to specify the exact width of the integer in bits. These are `int8_t` or `uint8_t` for 8-bit signed and unsigned integers, respectively. You may also use 16-bit (`int16_t`, `uint16_t`), 32-bit (`int32_t`, `uint32_t`), and 64-bit (`int64_t`, `uint64_t`) integer types. To use these types, remember to include header `<cstdint>`.

Avoid Narrowing Conversion Errors by Using List Initialization

When you initialize a variable of a smaller integer type (say, `short`) using another of a larger type (say, an `int`), you are risking a narrowing conversion error, because the compiler has to fit data stored in a type that can potentially hold much larger numbers into a type that doesn't have the same capacity (that is, is narrower). Here's an example:

```
int largeNum = 5000000;
short smallNum = largeNum; // compiles OK, yet narrowing error
```

Narrowing isn't restricted to conversions between integer types only. You may face narrowing errors if you initialize a `float` using a `double`, a `float` (or `double`) using an `int`, or an `int` using a `float`. Some compilers may warn, but this warning will not cause an error that stops compilation. In such cases, you may be confronted by bugs that occur infrequently and at execution time.

To avoid this problem, C++11 recommends *list initialization* techniques that prevent narrowing. To use this feature, insert initialization values/variables within braces {…}. The list initialization syntax is as follows:

```
int largeNum = 5000000;
short anotherNum{ largeNum }; // error! Amend types
int anotherNum{ largeNum }; // OK!
float someFloat{ largeNum }; // error! An int may be narrowed
float someFloat{ 5000000 }; // OK! 5000000 can be accomodated
```

It may not be immediately apparent, but this feature has the potential to spare bugs that occur when data stored in a type undergoes a narrowing conversion at execution time—these occur implicitly during an initialization and are tough to solve.

Automatic Type Inference Using `auto`

There are cases where the type of a variable is apparent given the initialization value being assigned to it. For example, if a variable is being initialized with the value `true`, the type of the variable can be best estimated as `bool`. Compilers supporting C++11 and beyond give you the option of not having to explicitly specify the variable type when using the keyword `auto`.

```
auto coinFlippedHeads = true;
```

We have left the task of defining an exact type for variable coinFlippedHeads to the compiler. The compiler checks the nature of the value the variable is being initialized to and then decides on the best possible type that suits this variable. In this particular case, it is clear that an initialization value of true best suits a variable that is of type bool. The compiler thus determines bool as the type that suits variable coinFlippedHeads best and internally treats coinFlippedHeads as a bool, as also demonstrated by Listing 3.6.

LISTING 3.6 Using the auto Keyword and Relying on the Compiler's Type-Inference Capabilities

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     auto coinFlippedHeads = true;
 7:     auto largeNumber = 2500000000000;
 8:
 9:     cout << "coinFlippedHeads = " << coinFlippedHeads;
10:     cout << " , sizeof(coinFlippedHeads) = " << sizeof(coinFlippedHeads) <<
endl;
11:     cout << "largeNumber = " << largeNumber;
12:     cout << " , sizeof(largeNumber) = " << sizeof(largeNumber) << endl;
13:
14:     return 0;
15: }
```

Output ▼

```
coinFlippedHeads = 1 , sizeof(coinFlippedHeads) = 1
largeNumber = 2500000000000 , sizeof(largeNumber) = 8
```

Analysis ▼

See how instead of deciding that coinFlippedHeads should be of type bool or that largeNumber should be a long long, you have used the auto keyword in Lines 6 and 7 where the two variables have been declared. This delegates the decision on the type of variable to the compiler, which uses the initialization value as a ballpark. You have used sizeof to actually check whether the compiler created the types you suspected it would, and you can check against the output produced by your code to verify that it really did.

NOTE

> Using `auto` requires you to initialize the variable for the compiler that uses this initial value in deciding what the variable type can be.
>
> When you don't initialize a variable of type `auto`, you get a compile error.

Even if `auto` seems to be a trivial feature at first sight, it makes programming a lot easier in those cases where the type variable is a complex type. The role of `auto` in writing simpler, yet type-safe code is revisited in Lesson 15, "An Introduction to the Standard Template Library," and beyond.

Using `typedef` to Substitute a Variable's Type

C++ allows you to substitute variable types to something that you might find convenient. You use the keyword `typedef` for that. Here is an example where a programmer wants to call an `unsigned int` a descriptive `STRICTLY_POSITIVE_INTEGER`.

```
typedef unsigned int STRICTLY_POSITIVE_INTEGER;
STRICTLY_POSITIVE_INTEGER numEggsInBasket = 4532;
```

When compiled, the first line tells the compiler that a `STRICTLY_POSITIVE_INTEGER` is nothing but an unsigned int. At later stages when the compiler encounters the already defined type `STRICTLY_POSITIVE_INTEGER`, it substitutes it for unsigned `int` and continues compilation.

NOTE

> `typedef` or type substitution is particularly convenient when dealing with complex types that can have a cumbersome syntax, for example, types that use templates. Templates are discussed later in Lesson 14, "An Introduction to Macros and Templates."

What Is a Constant?

Imagine you are writing a program to calculate the area and the circumference of a circle. The formulas are

```
Area = pi * Radius * Radius;
Circumference = 2 * pi * Radius
```

In this formula, `pi` is the constant of value 22 / 7. You don't want the value of `pi` to change anywhere in your program. You also want to avoid any accidental assignments of possibly incorrect values to `pi`. C++ enables you to define `pi` as a constant that cannot be changed after declaration. In other words, after it's defined, the value of a constant cannot be altered. Assignments to a constant in C++ cause compilation errors.

Thus, constants are like variables in C++ except that these cannot be changed. Similar to variables, constants also occupy space in the memory and have a name to identify the address where the space is reserved. However, the content of this space cannot be overwritten. Constants in C++ can be

- Literal constants
- Declared constants using the `const` keyword
- Constant expressions using the `constexpr` keyword (new since C++11)
- Enumerated constants using the `enum` keyword
- Defined constants that are not recommended and deprecated

Literal Constants

Literal constants can be of many types—integer, string, and so on. In your first C++ program in Listing 1.1, you displayed "Hello World" using the following statement:

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

In here, "Hello World" is a string literal constant. You literally have been using literal constants all the while! When you declare an integer `someNumber`, like this:

```
int someNumber = 10;
```

The integer variable `someNumber` is assigned an initial value of ten. Here decimal ten is a part of the code, gets compiled into the application, is unchangeable, and is a literal constant too. You may initialize the integer using a literal in octal notation, like this:

```
int someNumber = 012 // octal 12 evaluates to decimal 10
```

Starting in C++14, you may also use binary literals, like this:

```
int someNumber = 0b1010; // binary 1010 evaluates to decimal 10
```

TIP

C++ also allows you to define your own literals. For example, temperature as `32.0_F` (Fahrenheit) or `0.0_C` (Centigrade), distance as `16_m` (Miles) or `10_km` (Kilometers), and so on.

These suffixes `_F`, `_C`, `_m`, and `_km` are called user-defined literals and are explained in Lesson 12, "Operator Types and Operator Overloading," after the prerequisite concepts are explained.

Declaring Variables as Constants Using `const`

The most important type of constants in C++ from a practical and programmatic point of view are declared by using keyword `const` before the variable type. The generic declaration looks like the following:

```
const type-name constant-name = value;
```

Let's see a simple application that displays the value of a constant called `pi` (see Listing 3.7).

LISTING 3.7 Declaring a Constant Called `pi`

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     using namespace std;
 6:
 7:     const double pi = 22.0 / 7;
 8:     cout << "The value of constant pi is: " << pi << endl;
 9:
10:     // Uncomment next line to view compile failure
11:     // pi = 345;
12:
13:     return 0;
14: }
```

Output ▼

```
The value of constant pi is: 3.14286
```

Analysis ▼

Note the declaration of constant `pi` in Line 7. We use the `const` keyword to tell the compiler that `pi` is a constant of type `double`. If you uncomment Line 11 where the

programmer tries to assign a value to a variable you have defined as a constant, you see a compile failure that says something similar to, "You cannot assign to a variable that is `const`." Thus, constants are a powerful way to ensure that certain data cannot be modified.

NOTE

It is good programming practice to define variables that are not supposed to change their values as `const`. The usage of the `const` keyword indicates that the programmer has thought about ensuring the constantness of data where required and protects his application from inadvertent changes to this constant.

This is particularly useful in a multiprogrammer environment.

Constants are useful when declaring the length of static arrays, which are fixed at compile time. Listing 4.2 in Lesson 4, "Managing Arrays and Strings," includes a sample that demonstrates the use of a `const int` to define the length of an array.

Constant Expressions Using `constexpr`

Keyword `constexpr` allows function-like declaration of constants:

```
constexpr double GetPi() {return 22.0 / 7;}
```

One `constexpr` can use another:

```
constexpr double TwicePi() {return 2 * GetPi();}
```

`constexpr` may look like a function, however, allows for optimization possibilities from the compiler's and application's point of view. So long as a compiler is capable of evaluating a constant expression to a constant, it can be used in statements and expressions at places where a constant is expected. In the preceding example, `TwicePi()` is a `constexpr` that uses a constant expression `GetPi()`. This will possibly trigger a compile-time optimization wherein every usage of `TwicePi()` is simply replaced by `6.28571` by the compiler, and not the code that would calculate `2 x 22 / 7` when executed.

Listing 3.8 demonstrates the usage of `constexpr`.

LISTING 3.8 Using `constexpr` to Calculate Pi

```
1: #include <iostream>
2: constexpr double GetPi() { return 22.0 / 7; }
3: constexpr double TwicePi() { return 2 * GetPi(); }
4:
5: int main()
```

```
 6: {
 7:     using namespace std;
 8:     const double pi = 22.0 / 7;
 9:
10:     cout << "constant pi contains value " << pi << endl;
11:     cout << "constexpr GetPi() returns value " << GetPi() << endl;
12:     cout << "constexpr TwicePi() returns value " << TwicePi() << endl;

13:     return 0;
14: }
```

Output ▼

```
constant pi contains value 3.14286
constexpr GetPi() returns value 3.14286
constexpr TwicePi() returns value 6.28571
```

Analysis ▼

The program demonstrates two methods of deriving the value of `pi`—one as a constant variable `pi` as declared in Line 8 and another as a constant expression `GetPi()` declared in Line 2. `GetPi()` and `TwicePi()` may look like functions, but they are not exactly. Functions are invoked at program execution time. But, these are constant expressions and the compiler had already substituted every usage of `GetPi()` by `3.14286` and every usage of `TwicePi()` by `6.28571`. Compile-time resolution of `TwicePi()` increases the speed of program execution when compared to the same calculation being contained in a function.

NOTE

Constant expressions need to contain simple implementations that return simple types like integer, double, and so on. C++14 allows `constexpr` to contain decision-making constructs such as if and switch statements. These conditional statements are discussed in detail in Lesson 6, "Controlling Program Flow."

The usage of `constexpr` will not guarantee compile-time optimization—for example, if you use a `constexpr` expression to double a user provided number. The outcome of such an expression cannot be calculated by the compiler, which may ignore the usage of `constexpr` and compile as a regular function.

To see a demonstration of how a constant expression is used in places where the compiler expects a constant, see the code sample in Listing 4.2 in Lesson 4.

TIP

In the previous code samples, we defined our own constant `pi` as an exercise in learning the syntax of declaring constants and `constexpr`. Yet, most popular C++ compilers already supply you with a reasonably precise value of `pi` in the constant `M_PI`. You may use this constant in your programs after including header file `<cmath>`.

Enumerations

There are situations where a particular variable should be allowed to accept only a certain set of values. These are situations where you don't want the colors in the rainbow to contain Turquoise or the directions on a compass to contain Left. In both these cases, you need a type of variable whose values are restricted to a certain set defined by you. *Enumerations* are exactly the tool you need in this situation and are characterized by the keyword `enum`. Enumerations comprise a set of constants called *enumerators*.

In the following example, the enumeration `RainbowColors` contains individual colors such as `Violet` as enumerators:

```
enum RainbowColors
{
   Violet = 0,
   Indigo,
   Blue,
   Green,
   Yellow,
   Orange,
   Red
};
```

Here's another enumeration for the cardinal directions:

```
enum CardinalDirections
{
   North,
   South,
   East,
   West
};
```

Enumerations are used as user-defined types. Variables of this type can be assigned a range of values restricted to the enumerators contained in the enumeration. So, if defining a variable that contains the colors of a rainbow, you declare the variable like this:

```
RainbowColors MyFavoriteColor = Blue; // Initial value
```

In the preceding line of code, you declared an enumerated constant `MyFavoriteColor` of type `RainbowColors`. This enumerated constant variable is restricted to contain any of the legal VIBGYOR colors and no other value.

NOTE

The compiler converts the enumerator such as `Violet` and so on into integers. Each enumerated value specified is one more than the previous value. You have the choice of specifying a starting value, and if this is not specified, the compiler takes it as 0. So, `North` is evaluated as value 0.

If you want, you can also specify an explicit value against each of the enumerated constants by initializing them.

Listing 3.9 demonstrates how enumerated constants are used to hold the four cardinal directions, with an initializing value supplied to the first one.

LISTING 3.9 Using Enumerated Values to Indicate Cardinal Wind Directions

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: enum CardinalDirections
 5: {
 6:    North = 25,
 7:    South,
 8:    East,
 9:    West
10: };
11:
12: int main()
13: {
14:     cout << "Displaying directions and their symbolic values" << endl;
15:     cout << "North: " << North << endl;
16:     cout << "South: " << South << endl;
17:     cout << "East: " << East << endl;
18:     cout << "West: " << West << endl;
19:
20:    CardinalDirections windDirection = South;
21:    cout << "Variable windDirection = " << windDirection << endl;
22:
23:    return 0;
24: }
```

Output ▼

```
Displaying directions and their symbolic values
North: 25
South: 26
East: 27
West: 28
Variable windDirection = 26
```

Analysis ▼

Note how we have enumerated the four cardinal directions but have given the first `North` an initial value of 25 (see Line 6). This automatically ensures that the following constants are assigned values 26, 27, and 28 by the compiler as demonstrated in the output. In Line 20 you create a variable of type `CardinalDirections` that is assigned an initial value `South`. When displayed on the screen in Line 21, the compiler dispatches the integer value associated with South, which is 26.

TIP

> You may want to take a look at Listings 6.4 and 6.5 in Lesson 6. They use `enum` to enumerate the days of the week and conditional processing to tell what the day of the user's choosing is named after.

Defining Constants Using `#define`

First and foremost, don't use this if you are writing a program anew. The only reason this book analyzes the definition of constants using `#define` is to help you understand certain legacy programs that do define constants such as `pi` using this syntax:

```
#define pi 3.14286
```

`#define` is a preprocessor macro, and what is done here is that all mentions of `pi` henceforth are replaced by 3.14286 for the compiler to process. Note that this is a text replacement (read: non-intelligent replacement) done by the preprocessor. The compiler neither knows nor cares about the actual type of the constant in question.

CAUTION

> Defining constants using the preprocessor via `#define` is deprecated and should not be used.

Keywords You Cannot Use as Variable or Constant Names

Some words are reserved by C++, and you cannot use them as variable names. These keywords have special meaning to the C++ compiler. Keywords include `if`, `while`, `for`, and `main`. A list of keywords defined by C++ is presented in Table 3.2 as well as in Appendix B, "C++ Keywords." Your compiler might have additional reserved words, so you should check its manual for a complete list.

TABLE 3.2 Major C++ Keywords

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
constexpr	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

In addition, the following words are reserved:

and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

DO	DON'T
DO give variables descriptive names, even if that makes them long.	**DON'T** give names that are too short or contain just a character.
DO initialize variables, and use list initialization to avoid narrowing conversion errors.	**DON'T** give names that use exotic acronyms known only to you.
DO ensure that the name of the variable explains its purpose.	**DON'T** give names that are reserved C++ keywords as these won't compile.
DO put yourself into the shoes of one who hasn't seen your code yet and think whether the name would make sense to him or her.	
DO check whether your team is following certain naming conventions and follow them.	

Summary

In this lesson you learned about using memory to store values temporarily in variables and constants. You learned that variables have a size determined by their type and that the operator `sizeof` can be used to determine the size of one. You got to know of different types of variables such as `bool`, `int`, and so on and that they are to be used to contain different types of data. The right choice of a variable type is important in effective programming, and the choice of a variable that's too small for the purpose can result in a wrapping error or an overflow situation. You learned about the keyword `auto`, where you let the compiler decide the data-type for you on the basis of the initialization value of the variable.

You also learned about the different types of constants and usage of the most important ones among them using the keywords `const`, `constexpr`, and `enum`.

Q&A

Q Why define constants at all if you can use regular variables instead of them?

A Constants, especially those declared using the keyword `const`, are your way of telling the compiler that the value of a particular variable be fixed and not allowed to change. Consequently, the compiler always ensures that the constant variable is never assigned another value, not even if another programmer was to take up your work and inadvertently try to overwrite the value. So, declaring constants where

you know the value of a variable should not change is a good programming practice and increases the quality of your application.

Q Why should I initialize the value of a variable?

A If you don't initialize, you don't know what the variable contains for a starting value. The starting value is just the contents of the location in the memory that are reserved for the variable. Initialization such as that seen here:

```
int myFavoriteNumber = 0;
```

writes the initial value of your choosing, in this case 0, to the memory location reserved for the variable `myFavoriteNumber` as soon as it is created. There are situations where you do conditional processing depending on the value of a variable (often checked against nonzero). Such logic does not work reliably without initialization because an unassigned or initiated variable contains junk that is often nonzero and random.

Q Why does C++ give me the option of using `short int` and `int` and `long int`? Why not just always use the integer that always allows for the highest number to be stored within?

A C++ is a programming language that is used to program for a variety of applications, many running on devices with little computing capacity or memory resources. The simple old cell phone is one example where processing capacity and available memory are both limited. In this case, the programmer can often save memory or speed or both by choosing the right kind of variable if he doesn't need high values. If you are programming on a regular desktop or a high-end smartphone, chances are that the performance gained or memory saved in choosing one integer type over another is going to be insignificant and in some cases even absent.

Q Why should I not use global variables frequently? Isn't it true that they're usable throughout my application and I can save some time otherwise lost to passing values around functions?

A Global variables can be read and assigned globally. The latter is the problem as they can be changed globally. Assume you are working on a project with a few other programmers in a team. You have declared your integers and other variables to be global. If any programmer in your team changes the value of your integer inadvertently in his code—which even might be a different .CPP file than the one you are using—the reliability of your code is affected. So, sparing a few seconds or minutes should not be criteria, and you should not use global variables indiscriminately to ensure the stability of your code.

Q C++ is giving me the option of declaring unsigned integers that are supposed to contain only positive integer values and zero. What happens if I decrement a zero value contained in an `unsigned int`?

A You see a wrapping effect. Decrementing an unsigned integer that contains 0 by 1 means that it wraps to the highest value it can hold! Check Table 3.1—you see that an `unsigned short` can contain values from 0 to 65,535. So, declare an `unsigned short` and decrement it to see the unexpected:

```
unsigned short myShortInt = 0;  // Initial Value
myShortInt = myShortInt - 1; // Decrement by 1
std::cout << myShortInt << std::endl; // Output: 65535!
```

Note that this is not a problem with the `unsigned short`, rather with your usage of the same. An unsigned integer (or short or long) is not to be used when negative values are within the specifications. If the contents of `myShortInt` are to be used to dynamically allocate those many number of bytes, a little bug that allows a zero value to be decremented would result in 64KB being allocated! Worse, if `myShortInt` were to be used as an index in accessing a location of memory, chances are high that your application would access an external location and would crash!

Workshop

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

Quiz

1. What is the difference between a signed and an unsigned integer?
2. Why should you not use `#define` to declare a constant?
3. Why would you initialize a variable?
4. Consider the `enum` below. What is the value of `Queen`?

```
enum YourCards {Ace, Jack, Queen, King};
```

5. What is wrong with this variable name?

```
int Integer = 0;
```

Exercises

1. Modify `enum YourCards` in quiz question 4 to demonstrate that the value of `Queen` can be 45.
2. Write a program that demonstrates that the size of an unsigned integer and a normal integer are the same, and that both are smaller in size than a long integer.
3. Write a program to calculate the area and circumference of a circle where the radius is fed by the user.
4. In Exercise 3, if the area and circumference were to be stored in integers, how would the output be any different?
5. **BUGBUSTERS:** What is wrong in the following initialization:

```
auto Integer;
```

Index

Symbols

A

B

C

D

E

F

J-K-L

M

Q-R

S

T

U

V

W-X-Y-Z