Jesse Liberty
Rogers Cadenhead

FIFTH EDITION
Starter Kit
CD includes
a fully-functional
C++ compiler

Sams Teach Yourself C++ in 24 Hours

SAMS

Jesse Liberty
Rogers Cadenhead

Sams Teach Yourself C++ in 24 Hours

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

Sams Teach Yourself C++ in 24 Hours

ISBN-13: 978-0-672-33331-6
ISBN-10: 0-672-33331-7

Printed in the United States of America

First Printing April 2011

Library of Congress Cataloging-in-Publication data is on file.

Trademarks

Warning and Disclaimer

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Editor in Chief
Mark Taub

Acquisitions Editor
Mark Taber

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Keith Cline

Indexer
Lisa Stumpf

Proofreader
Leslie Joseph

Technical Editor
Jon Upchurch

Publishing Coordinator
Vanessa Evans

Media Producer
Dan Scherf

Designer
Gary Adair

Compositor
Mark Shirar

Table of Contents

About the Authors

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

Rogers Cadenhead is a writer, computer programmer, and web developer who has written 23 books on Internet-related topics, including *Sams Teach Yourself Java in 21 Days* and *Sams Teach Yourself Java in 24 Hours*. He publishes the Drudge Retort and other websites that receive more than 22 million visits a year. This book's official website is at http://cplusplus.cadenhead.org.

Dedications

This book is dedicated to Edythe, who provided life; Stacey, who shares it; and Robin and Rachel, who give it purpose.

—Jesse Liberty

This book is dedicated to my dad, who's currently teaching himself something a lot harder than computer programming: how to walk again after spinal surgery. Through the many months of rehab, you've been an inspiration. I've never known someone with as much indefatigable determination to fix the hitch in his giddy-up.

—Rogers Cadenhead

Acknowledgments

With each book, there is a chance to acknowledge and to thank those folks without whose support and help this book literally would have been impossible. First among them are Stacey, Robin, and Rachel Liberty.

—Jesse Liberty

A book like this requires the hard work and dedication of numerous people. Most of them are at Sams Publishing in Indianapolis, and to them I owe considerable thanks—in particular, to Keith Cline, Mandie Frank, Songlin Qiu, Mark Taber, and Jon Upchurch. Most of all, I thank my incredible wife, Mary, and sons, Max, Eli, and Sam.

—Rogers Cadenhead

We Want to Hear from You!

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

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

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

When you write, please be sure to include this book's title and author as well as your name and contact information.

Email: feedback@samspublishing.com

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

Reader Services

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

This page intentionally left blank

Introduction

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

If you continue for another 23 hours, 59 minutes, and 40 seconds, you will master the fundamentals of the C++ programming language. Twenty-four 1-hour lessons cover the fundamentals, such as managing I/O, creating loops and arrays, using object-oriented programming with templates, and creating C++ programs.

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

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

Who Should Read This Book?

You don't need any previous experience in programming to learn C++ with this book.

This book starts with the basics and teaches you both the language and the concepts involved with programming C++. Whether you are just beginning or already have some experience programming, you will find that this book makes learning C++ fast and easy.

Should I Learn C First?

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

Why Should I Learn C++?

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

In spite of being created in 1979, C++ is still being used for professional software today because of the power and flexibility of the language. There's even a new version of the language coming up, which has the working title C++0x and makes the language even more useful.

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

What If I Don't Want This Book?

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

Conventions Used in This Book

This book contains special elements as described here.

By the Way

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

Watch Out!

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

Did you Know?

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

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

This book uses various typefaces:

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

This page intentionally left blank

HOUR 3

Creating Variables and Constants

What You'll Learn in This Hour:

- How to create variables and constants
- How to assign values to variables and change those values
- How to display the value of variables
- How to find out how much memory a variable requires

What Is a Variable?

A *variable* is a location in computer memory where you can store and retrieve a value. Your computer's memory can be thought of as a series of cubbyholes lined up in a long row. Each cubbyhole is numbered sequentially. The number of each cubbyhole is its memory address.

Variables have addresses and are given names that describe their purpose. In a game program, you could create a variable named `score` to hold the player's score and a variable named `zombies` for the number of zombies the player has defeated. A variable is a label on a cubbyhole so that it can be accessed without knowing the actual memory address.

Figure 3.1 shows seven cubbyholes with addresses ranging from 101 to 107. In address 104, the `zombies` variable holds the value 17. The other cubbyholes are empty.

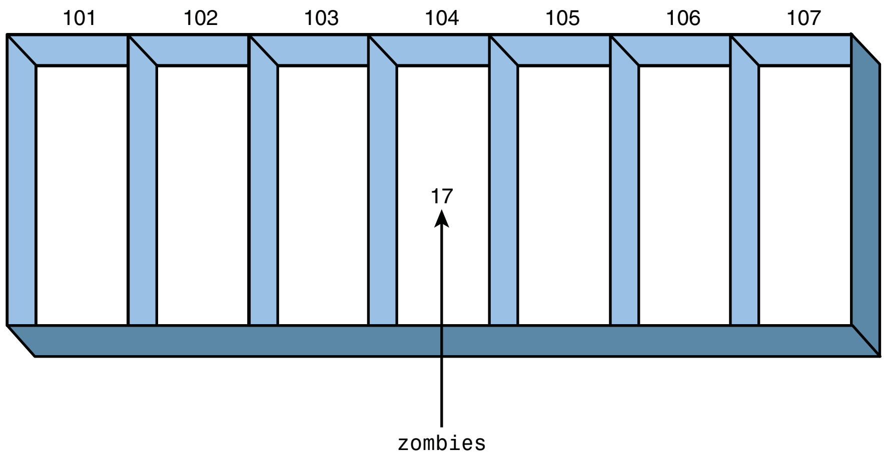

FIGURE 3.1
A visual representation of memory.

Storing Variables in Memory

When you create a variable in C++, you must tell the compiler the variable's name and what kind of information it will hold, such as an integer, character, or floating-point number. This is the variable's *type* (sometimes called *data type*). The type tells the compiler how much room to set aside in memory to hold the variable's value.

Each cubbyhole in memory can hold 1 byte. If a variable's type is 2 bytes in size, it needs 2 bytes of memory. Because computers use bytes to represent values, it is important that you familiarize yourself with this concept.

A short integer, represented by `short` in C++, is usually 2 bytes. A long integer (`long`) is 4 bytes, an integer (`int`) can be 2 or 4 bytes, and a long long integer is 8 bytes.

Characters of text are represented by the `char` type in C++, which usually is 1 byte in size. In Figure 3.1 shown earlier, each cubbyhole holds 1 byte. A single short integer could be stored in addresses 106 and 107.

True-false values are stored as the `bool` type. The values `true` and `false` are the only values it can hold.

The size of a `short` always is smaller than or the same as an `int`. The size of an `int` is always the same or smaller than a `long`. Floating-point numeric types are different and are discussed later this hour.

The usual type sizes thus far described do not hold true on all systems. You can check the size a type holds in C++ using `sizeof()`, an element of the language called a *function*. The parentheses that follow `sizeof` should be filled with the name of a type, as in this statement:

```
std:cout << sizeof(int) << "\n";
```

This statement displays the number of bytes required to store an integer variable. The `sizeof()` function is provided by the compiler and does not require an `include` directive. The Sizer program in Listing 3.1 relies on the `sizeof()` function to report the sizes of common C++ types on your computer.

LISTING 3.1 The Full Text of `Sizer.cpp`

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     std::cout << "The size of an integer:\t\t";
 6:     std::cout << sizeof(int) << " bytes\n";
 7:     std::cout << "The size of a short integer:\t";
 8:     std::cout << sizeof(short) << " bytes\n";
 9:     std::cout << "The size of a long integer:\t";
10:     std::cout << sizeof(long) << " bytes\n";
11:     std::cout << "The size of a character:\t";
12:     std::cout << sizeof(char) << " bytes\n";
13:     std::cout << "The size of a boolean:\t\t";
14:     std::cout << sizeof(bool) << " bytes\n";
15:     std::cout << "The size of a float:\t\t";
16:     std::cout << sizeof(float) << " bytes\n";
17:     std::cout << "The size of a double float:\t";
18:     std::cout << sizeof(double) << " bytes\n";
19:     std::cout << "The size of a long long int:\t";
20:     std::cout << sizeof(long long int) << " bytes\n";
21:
22:     return 0;
23: }
```

This program makes use of a new feature of C++0x, the next version of the language. The `long long int` data type holds extremely large integers. If your compiler fails with an error, it may not support this feature yet. Delete lines 19–20 and try again to see if that's the problem.

After being compiled, this program produces the following output when run on a Linux Ubuntu 9.10 system:

```
The size of an integer:         4 bytes
The size of a short integer:    2 bytes
The size of a long integer:     4 bytes
The size of a character:        1 bytes
The size of a boolean:          1 bytes
The size of a float:            4 bytes
The size of a double float:     8 bytes
The size of a long long int:    8 bytes
```

Compare this output to how it runs on your computer. The `sizeof()` function reveals the size of an object specified as its argument. For example, on line 16 the keyword `float` is passed to `sizeof()`. As you can see from the output, on the Ubuntu computer an `int` is equivalent in size to a `long`.

Signed and Unsigned Variables

All the integer types come in two varieties specified using a keyword. They are declared with `unsigned` when they only hold positive values and `signed` when they hold positive or negative values. Here's a statement that creates a short `int` variable called `zombies` that does not hold negative numbers:

```
unsigned short zombies = 0;
```

The variable is assigned the initial value 0. Both signed and unsigned integers can equal 0.

Integers that do not specify either `signed` or `unsigned` are assumed to be signed.

Signed and unsigned integers are stored using the same number of bytes. For this reason, the largest number that can be stored in an unsigned integer is twice as big as the largest positive number that a signed integer can hold. An `unsigned short` can handle numbers from 0 to 65,535. Half the numbers represented by a `signed short` are negative, so a `signed short` represents numbers from –32,768 to 32,767. In both cases, the total number of possible values is 65,535.

Variable Types

In addition to integer variables, C++ types cover floating-point values and characters of text.

Floating-point variables have values that can be expressed as decimal values. Character variables hold a single byte representing 1 of the 256 characters and symbols in the standard ASCII character set.

Variable types supported by C++ programs are shown in Table 3.1, which lists the variable type, the most common memory size, and the possible values that it can hold. Compare this table to the output of the Sizer program when run on your computer, looking for size differences.

TABLE 3.1 Variable Types

Type	Size	Values
`unsigned short`	2 bytes	0 to 65,535
`short`	2 bytes	–32,768 to 32,767
`unsigned long`	4 bytes	0 to 4,294,967,295
`long`	4 bytes	–2,147,483,648 to 2,147,483,647
`int`	4 bytes	–2,147,483,648 to 2,147,483,647

TABLE 3.1 Continued

Type	Size	Values
`unsigned int`	4 bytes	0 to 4,294,967,295
`long long int`	8 bytes	-9.2 quintillion to 9.2 quintillion
`char`	1 byte	256 character values
`bool`	1 byte	true or false
`float`	4 bytes	1.2e–38 to 3.4e38
`double`	8 bytes	2.2e–308 to 1.8e308

The `short` and `long` variables also are called `short int` and `long int` in C++. Both forms are acceptable in your programs.

As shown in Table 3.1, `unsigned short` integers can hold a value only up to 65,535, while `signed short` integers can hold half that at maximum. Although `unsigned long long int` integers can hold more than 18.4 quintillion, that's still finite. If you need a larger number, you must use `float` or `double` at the cost of some numeric precision. Floats and doubles can hold extremely large numbers, but only the first 7 or 19 digits are significant on most computers. Additional digits are rounded off.

Although it's considered poor programming practice, a `char` variable can be used as a very small integer. Each character has a numeric value equal to its ASCII code in that character set. For example, the exclamation point character (!) has the value 33.

Defining a Variable

A variable is defined in C++ by stating its type, the variable name, and a colon to end the statement, as in this example:

```
int highScore;
```

More than one variable can be defined in the same statement as long as they share the same type. The names of the variables should be separated by commas, as in these examples:

```
unsigned int highScore, playerScore;
long area, width, length;
```

The `highScore` and `playerScore` variables are both unsigned integers. The second statement creates three `long` integers: `area`, `width`, and `length`. Because these integers share the same type, they can be created in one statement.

A variable name can be any combination of uppercase and lowercase letters, numbers and underscore characters (_) without any spaces. Legal variable names include `x`, `driver8`, and `playerScore`. C++ is case sensitive, so the `highScore` variable differs from ones named `highscore` or `HIGHSCORE`.

Using descriptive variable names makes it easier to understand a program for the humans reading it. (The compiler doesn't care one way or the other.) Take a look at the following two code examples to see which one is easier to figure out.

Example 1

```
main()
{
     unsigned short x;
     unsigned short y;
     unsigned int z;
     z = x * y;
}
```

Example 2

```
main ()
{
     unsigned short width;
     unsigned short length;
     unsigned short area;
     area = width * length;
}
```

Programmers differ in the conventions they adopt for variable names. Some prefer all lowercase letters for variable names with underscores separating words, such as `high_score` and `player_score`. Others prefer lowercase letters except for the first letter of new words, such as `highScore` and `playerScore`. (In a bit of programming lore, the latter convention has been dubbed CamelCase because the middle-of-word capitalization looks like a camel's hump.)

Programmers who learned in a UNIX environment tend to use the first convention, whereas those in the Microsoft world use CamelCase. The compiler does not care.

The code in this book uses CamelCase.

With well-chosen variable names and plenty of comments, your C++ code will be much easier to figure out when you come back to it months or years later.

Watch Out!

> Some compilers allow you to turn case sensitivity of variable names off. Do not do this. If you do, your programs won't work with other compilers, and other C++ programmers will make fun of you.

Some words are reserved by C++ and may not be used as variable names because they are keywords used by the language. Reserved keywords include `if`, `while`, `for`, and `main`. Generally, any reasonable name for a variable is almost certainly not a keyword.

Variables may contain a keyword as part of a name but not the entire name, so variables `mainFlag` and `forward` are permitted but `main` and `for` are reserved.

Assigning Values to Variables

A variable is assigned a value using the = operator, which is called the *assignment operator*. The following statements show it in action to create an integer named `highScore` with the value 13,000:

```
unsigned int highScore;
highScore = 13000;
```

A variable can be assigned an initial value when it is created:

```
unsigned int highScore = 13000;
```

This is called *initializing the variable*. Initialization looks like assignment, but when you work later with constants, you'll see that some variables must be initialized because they cannot be assigned a value.

The Rectangle program in Listing 3.2 uses variables and assignments to compute the area of a rectangle and display the result.

LISTING 3.2 The Full Text of `Rectangle.cpp`

```
#include <iostream>

int main()
{
    // set up width and length
    unsigned short width = 5, length;
    length = 10;

    // create an unsigned short initialized with the
    // result of multiplying width by length
    unsigned short area = width * length;

    std::cout << "Width: " << width << "\n";
    std::cout << "Length: "  << length << "\n";
```

LISTING 3.2 Continued

```
15:     std::cout << "Area: " << area << "\n";
16:     return 0;
17: }
```

This program produces the following output when run:

```
Width: 5
Length: 10
Area: 50
```

Like the other programs you've written so far, Rectangle uses the `#include` directive to bring the standard `iostream` library into the program. This makes it possible to use `std::cout` to display information.

Within the program's `main()` block, on line 6 the variables `width` and `length` are created and `width` is given the initial value of 5. On line 7, the `length` variable is given the value 10 using the = assignment operator.

On line 11, an integer named `area` is defined. This variable is initialized with the value of the variable `width` multiplied by the value of `length`. The multiplication operator * multiplies one number by another.

On lines 13–15, the values of all three variables are displayed.

Using Type Definitions

When a C++ program contains a lot of variables, it can be repetitious and error-prone to keep writing `unsigned short int` for each one. A shortcut for an existing type can be created with the keyword `typedef`, which stands for type definition.

A `typedef` requires `typedef` followed by the existing type and its new name. Here's an example:

```
typedef unsigned short USHORT
```

This statement creates a type definition named `USHORT` that can be used anywhere in a program in place of `unsigned short`. The NewRectangle program in Listing 3.3 is a rewrite of Rectangle that uses this type definition.

LISTING 3.3 The Full Text of `NewRectangle.cpp`

```
#include <iostream>

int main()
{
    // create a type definition
    typedef unsigned short USHORT;

    // set up width and length
    USHORT width = 5;
```

```
10:     USHORT length = 10;
11:
12:     // create an unsigned short initialized with the
13:     // result of multiplying width by length
14:     USHORT area = width * length;
15:
16:     std::cout << "Width: " << width << "\n";
17:     std::cout << "Length: "  << length << "\n";
18:     std::cout << "Area: " << area << "\n";
19:     return 0;
20: }
```

This program has the same output as Rectangle: the values of `width` (5), `length` (10), and `area` (50).

On line 6, the `USHORT` typedef is created as a shortcut for `unsigned short`. A type definition substitutes the underlying definition `unsigned short` wherever the shortcut `USHORT` is used.

During Hour 8, “Creating Basic Classes,” you learn how to create new types in C++. This is a different from creating type definitions.

By the Way

> Some compilers will warn that in the Rectangle2 program a “conversion may lose significant digits.” This occurs because the product of the two `USHORTS` on line 14 might be larger than an unsigned short integer can hold. For this program, you can safely ignore the warning.

Constants

A constant, like a variable, is a memory location where a value can be stored. Unlike variables, constants never change in value. You must initialize a constant when it is created. C++ has two types of constants: literal and symbolic.

A literal constant is a value typed directly into your program wherever it is needed. For example, consider the following statement:

```
long width = 5;
```

This statement assigns the integer variable `width` the value 5. The 5 in the statement is a literal constant. You can’t assign a value to 5, and its value can’t be changed.

The values `true` and `false`, which are stored in `bool` variables, also are literal constants.

A symbolic constant is a constant represented by a name, just like a variable. The `const` keyword precedes the type, name, and initialization. Here’s a statement that sets the point reward for killing a zombie:

```
const int KILL_BONUS = 5000;
```

Whenever a zombie is dispatched, the player's score is increased by the reward:

```
playerScore = playerScore + KILL_BONUS;
```

If you decide later to increase the reward to 10,000 points, you can change the constant KILL_BONUS, and it will be reflected throughout the program. If you were to use the literal constant 5000 instead, it would be more difficult to find all the places it is used and change the value. This reduces the potential for error.

Well-named symbolic constants also make a program more understandable. Constants often are fully capitalized by programmers to make them distinct from variables. This is not required by C++, but the capitalization of a constant must be consistent because the language is case sensitive.

Defining Constants

There's another way to define constants that dates back to early versions of the C language, the precursor of C++. The preprocessor directive #define can create a constant by specifying its name and value, separated by spaces:

```
#define KILLBONUS 5000
```

The constant does not have a type such as int or char. The #define directive enables a simple text substitution that replaces every instance of KILLBONUS in the code with 5000. The compiler sees only the end result.

Because these constants lack a type, the compiler cannot ensure that the constant has a proper value.

Enumerated Constants

Enumerated constants create a set of constants with a single statement. They are defined with the keyword enum followed by a series of comma-separated names surrounded by braces:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

This statement creates a set of enumerated constants named COLOR with five values named RED, BLUE, GREEN, WHITE and BLACK.

The values of enumerated constants begin with 0 for the first in the set and count upwards by 1. So RED equals 0, BLUE equals 1, GREEN equals 2, WHITE equals 3, and BLACK equals 4. All the values are integers.

Constants also can specify their value using an = assignment operator:

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

This statement sets `RED` to 100, `GREEN` to 500, and `BLACK` to 700. The members of the set without assigned values will be 1 higher than the previous member, so `BLUE` equals 101 and `WHITE` equals 501.

The advantage of this technique is that you get to use a symbolic name such as `BLACK` or `WHITE` rather than a possibly meaningless number such as 1 or 700.

Summary

This hour covered how to work with simple kinds of information in C++ such as integers, floating-point values, and characters. Variables are used to store values that can change as a program runs. Constants store values that stay the same—in other words, they are not variable.

The biggest challenge when using variables is choosing the proper type. If you're working with signed integers that might go higher than 65,000, you should store them in a `long` rather than a `short`. If they might go higher than 2.1 billion, they're too big for a `long`. If a numeric value contains decimal values, it must be either `float` or `double`, the two floating-point types in the C++ language.

Another thing to keep in mind when working with variables is the number of bytes they occupy, which can vary on different systems. The `sizeof()` function provided by the compiler returns the number of bytes any variable type requires.

Q&A

Q. ***If a short int can run out of room, why not always use long integers?***

A. Both `short` integers and `long` integers will run out of room, but a `long` integer will do so with a much larger number. On most computers, a `long` integer takes up twice as much memory, which has become less of a concern because of the memory available on modern PCs.

Q. ***What happens if I assign a number with a decimal to an integer rather than a float or double? Consider the following line of code:***

```
int rating = 5.4;
```

A. Some compilers issue a warning, but the assignment of a decimal value to an integer type is permitted in C++. The number is truncated into an integer, so the statement assigns the `rating` integer the value 5. The more precise information is lost in the assignment, so if you tried to assign `rating` to a `float` variable, it would still equal 5.

Q. ***Why should I bother using symbolic constants?***

A. When a constant is used in several places in a program, a symbolic constant enables all the values to change simply by changing the constant's initialization. Symbolic constants also serve an explanatory purpose like comments. If a statement multiplies a number by 360, it's less easily understood than multiplying it by a constant named `degreesInACircle` that equals 360.

Q. ***Why did Jack Klugman have a 40-year feud with Norman Fell?***

A. Klugman, the star of the TV shows *Quincy M.E.* and *The Odd Couple*, had a well-publicized long-running spat with Fell, the star of *Three's Company* and the landlord on *The Graduate*. No one seems to know the cause, but it did not end until Fell's death in 1998.

The movie reference site IMDb quotes Fell as saying, "I could have killed as Oscar. I would have been great as Quincy. I wouldn't have been so hammy. Klugman overacted every scene. You want the show to be good, pick me. You want a chain-smoking jackass who ruins any credibility for your project, I'll give you Klugman's number."

IMDb quotes Klugman as saying after Fell's funeral, "Best funeral I've ever been to. I've never laughed so hard in years. I had the time of my life."

The two actors, born in Philadelphia two years apart, bear some resemblance to each other and could have competed for the same roles over the decades they were acting in films and television. In reality, however, they were not enemies. As the blogger Tom Nawrocki found out in 2008, their feud was a shared joke they played on the media.

Workshop

Now that you've learned about variables and constants, you can answer a few questions and do a couple of exercises to firm up your knowledge about them.

Quiz

1. Why would you use unsigned over signed integers?

   A. They hold more numbers.

   B. They hold more positive numbers.

   C. There's no reason to prefer one over the other.

2. Are the variables `ROSE`, `rose`, and `Rose` the same?

   A. Yes

   B. No

   C. None of your business

3. What is the difference between a `#define` constant and `const`?

   A. Only one is handled by the preprocessor.

   B. Only one has a type.

   C. Both a and b

Answers

1. B. Unsigned integers hold more positive values and cannot be used to hold negative values. They hold the same number of values.

2. B. Because C++ is case sensitive, a `ROSE` is not a `rose` is not a `Rose`. Each reference is treated as a different variable by the compiler.

3. C. The preprocessor directive `#define` substitutes the specified value into your code every place it appears in code. It does not have a data type and is invisible to the compiler. A constant, created with the keyword `const`, has a data type and is handled by the compiler.

Activities

1. Create a program that uses constants for a touchdown (6 points), field goal (3 points), extra point (1 point), and safety (2 point) and then adds them in the same order they were scored by the teams in the last Super Bowl. Display the final score. (For extra credit, make the Indianapolis Colts win.)

2. Expand the Rectangle program so that it determines the area of a three-dimensional rectangle that has width, length, and height. To determine the area, use the multiplication operator * to multiply all three values.

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

Index

Symbols

A

B

D

E

M

P-Q

R

V

W-X

Y-Z