Sams Teach Yourself

C++
in 24 Hours
# Table of Contents

**Introduction**  
1  

### Part I: Beginning C++

**HOUR 1: Writing Your First Program**  
5  
- Using C++ .......................................................... 5  
- Finding a Compiler ................................................. 6  
- Compiling and Linking the Source Code .................. 9  
- Creating Your First Program ................................. 10  

**HOUR 2: Organizing the Parts of a Program**  
15  
- Reasons to Use C++ .............................................. 15  
- The Parts of a Program ........................................... 19  
- Comments .......................................................... 22  
- Functions ......................................................... 23  

**HOUR 3: Creating Variables and Constants**  
29  
- What Is a Variable? .............................................. 29  
- Defining a Variable .............................................. 33  
- Assigning Values to Variables ............................... 35  
- Using Type Definitions ......................................... 36  
- Constants .......................................................... 37  

**HOUR 4: Using Expressions, Statements, and Operators**  
43  
- Statements ......................................................... 43  
- Expressions ....................................................... 44  
- Operators .......................................................... 45  
- If-Else Conditional Statements ............................ 53  
- Logical Operators .............................................. 56  
- Tricky Expression Values ...................................... 58
### HOUR 9: Moving into Advanced Classes

- const Member Functions ........................................... 125
- Interface Versus Implementation ................................... 126
- Organizing Class Declarations and Function Definitions ......................................................... 126
- Inline Implementation .................................................... 127
- Classes with Other Classes as Member Data ......................................................... 129

### Part III: Memory Management

#### HOUR 10: Creating Pointers

- Understanding Pointers and Their Usage ................................................ 137
- The Stack and the Heap ....................................................... 146

#### HOUR 11: Developing Advanced Pointers

- Creating Objects on the Heap .................................................. 155
- Deleting Objects .................................................................. 155
- Accessing Data Members Using Pointers ........................................ 157
- Member Data on the Heap ...................................................... 158
- The this Pointer ................................................................. 160
- Stray or Dangling Pointers ..................................................... 161
- const Pointers ....................................................................... 162
- const Pointers and const Member Functions ..................................... 163

#### HOUR 12: Creating References

- What Is a Reference? ......................................................... 169
- Creating a Reference .......................................................... 170
- Using the Address of Operator on References ........................................... 171
- What Can Be Referenced? .................................................... 173
- Null Pointers and Null References ........................................... 174
- Passing Function Arguments by Reference .......................................... 174
- Understanding Function Headers and Prototypes ....................................... 179
- Returning Multiple Values .................................................. 179
HOUR 13: Developing Advanced References and Pointers

- Passing by Reference for Efficiency ............................................. 185
- Passing a const Pointer ............................................................... 188
- References as an Alternative to Pointers .................................. 191
- When to Use References and When to Use Pointers ................. 192
- Don’t Return a Reference to an Object That Isn’t in Scope! ........ 193
- Returning a Reference to an Object on the Heap ...................... 194
- Pointer, Pointer, Who Has the Pointer? ................................. 196

Part IV: Advanced C++

HOUR 14: Calling Advanced Functions ........................................ 201
- Overloaded Member Functions ................................................... 201
- Using Default Values .................................................................. 203
- Initializing Objects ..................................................................... 205
- The Copy Constructor .................................................................. 206

HOUR 15: Using Operator Overloading ........................................ 215
- Operator Overloading ............................................................... 215
- Conversion Operators ............................................................... 225

Part V: Inheritance and Polymorphism

HOUR 16: Extending Classes with Inheritance ......................... 233
- What Is Inheritance? ................................................................. 233
- Private Versus Protected .......................................................... 236
- Constructors and Destructors ..................................................... 238
- Passing Arguments to Base Constructors ................................. 241
- Overriding Functions ............................................................... 245

HOUR 17: Using Polymorphism and Derived Classes ............... 253
- Polymorphism Implemented with Virtual Methods .................. 253
- How Virtual Member Functions Work .................................... 257
# Table of Contents

**HOUR 18: Making Use of Advanced Polymorphism** 269
- Problems with Single Inheritance .................................................. 269
- Abstract Data Types ....................................................................... 273

**HOUR 19: Storing Information in Linked Lists** 289
- Linked Lists and Other Structures .................................................. 289
- Linked List Case Study ................................................................... 290
- Linked Lists as Objects ................................................................... 299

**Part VI: Special Topics**

**HOUR 20: Using Special Classes, Functions, and Pointers** 303
- Static Member Data ......................................................................... 303
- Static Member Functions ............................................................. 305
- Containment of Classes .................................................................. 307
- Friend Classes and Functions ...................................................... 313

**HOUR 21: Using New Features of C++0x** 331
- The Next Version of C++ ............................................................. 331
- Null Pointer Constant ..................................................................... 332
- Compile-Time Constant Expressions ............................................ 333
- Auto-Typed Variables .................................................................... 335
- New for Loop ................................................................................. 338

**HOUR 22: Employing Object-Oriented Analysis and Design** 343
- The Development Cycle .................................................................. 343
- Simulating an Alarm System ........................................................ 344
- PostMaster: A Case Study ............................................................. 351

**HOUR 23: Creating Templates** 373
- What Are Templates? ........................................................................ 373
- Instances of the Template ............................................................ 374
- Template Definition ....................................................................... 374
- Using Template Items ...................................................................... 381
**HOUR 24: Dealing with Exceptions and Error Handling**

- Bugs, Errors, Mistakes, and Code Rot .................................................. 389
- Handling the Unexpected ........................................................................ 390
- Exceptions .......................................................................................... 391
- Using try and catch Blocks ................................................................. 395
- Writing Professional-Quality Code ...................................................... 400

**Part VII: Appendices**

- **APPENDIX A: Binary and Hexadecimal** ............................................. 409
  - Other Bases .................................................................................. 410
  - Around the Bases ........................................................................ 410
  - Hexadecimal ................................................................................. 414

- **APPENDIX B: Glossary** ................................................................. 419

- **APPENDIX C: This Book’s Website** ................................................. 427

Index .................................................................................................. 429
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@sams publishing.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
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.
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";
```
What Is a Variable?

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**

```cpp
#include <iostream>

int main()
{
    std::cout << "The size of an integer: \t\t";
    std::cout << sizeof(int) << " bytes\n";
    std::cout << "The size of a short integer: \t";
    std::cout << sizeof(short) << " bytes\n";
    std::cout << "The size of a long integer: \t";
    std::cout << sizeof(long) << " bytes\n";
    std::cout << "The size of a character: \t";
    std::cout << sizeof(char) << " bytes\n";
    std::cout << "The size of a boolean: \t\t";
    std::cout << sizeof(bool) << " bytes\n";
    std::cout << "The size of a float: \t\t";
    std::cout << sizeof(float) << " bytes\n";
    std::cout << "The size of a double float: \t";
    std::cout << sizeof(double) << " bytes\n";
    std::cout << "The size of a long long int: \t";
    std::cout << sizeof(long long int) << " bytes\n";
    return 0;
}
```

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 byte
The size of a boolean: 1 byte
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:

```c
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>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>–32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
<td>–2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>–2,147,483,648 to 2,147,483,647</td>
</tr>
</tbody>
</table>
TABLE 3.1

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

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:

```cpp
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:

```c++
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**

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

**Example 2**

```c++
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.

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:

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

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

```cpp
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`

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

```cpp
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

```cpp
#include <iostream>

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

    // set up width and length
    USHORT width = 5;
```
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 different from creating type definitions.

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:

```cpp
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:

```cpp
const int KILL_BONUS = 5000;
```
Whenever a zombie is dispatched, the player's score is increased by the reward:

```cpp
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:

```cpp
#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:

```cpp
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:

```cpp
eenum 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:*

```cpp
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.
Symbols

#, 26
#include preprocessor, 26
%, 46
() parentheses, 51, 59, 157
*/ (star-slash comments), 22, 26
/* (slash-star comments), 22, 26
// (double slash comments), 22, 26
* (asterisk), 171
* (dereference operator), 142
+ operator, 223
++ (plus-plus), 47
-- (minus-minus), 47
. (dot operator), 114
80/80 rule, PostMaster, 358-359

= (assignment operator), 35
== (equality operator), 52, 225
>passing by references, 185

A
abstract data types, 273-276
hierarchies, 280-284
virtual functions, 276-277
implementing, 277-280
abstraction, hierarchies, 280-284
abstracts, types, 284
access in code, 405
accessing
class members, 114
data members with
pointers, 157-158
members of contained
class, 312
accessors

accessors, 115
add() function, 25
adding
classes, simulating alarm systems, 348
variables in C++, 49
addition operator, overloading, 220-222
Addresser.cpp, 138
addresses, 143
of operators on references, 171-173
pointers, 144-146
storing in pointers, 140-142
ADT (abstract data types), virtual functions, 276
advanced for loops, 88
Alabama Crimson Tide, 407
alarm systems, simulating, 344
adding more classes, 348
analysis and requirements, 345
conceptualization, 345
designing classes, 347-348
event loops, 348-350
high-level and low-level design, 346
analysis, simulating alarm systems, 345
Animal.cpp, 280-283
APIs (application programming interfaces), 359
PostMaster, 359-361
Area.cpp, 65, 73
AreaCube program, 73
arguments, 23
functions, 24-25
passing to base constructors, 241-245
ArrayFunction.cpp, 317-319
arrays, 97-99
accessing, 99
character, 104-106
initializing, 100
multidimensional, 101-102
initializing, 102-103
memory, 103
of pointers to functions, 317-319
peaks, 97
writing past the end of, 99
assigning values to variables, 35-36
assignment operator, 35, 229
assignment operators, 46
Assignment.cpp, 172,
  224-225
asterisk (*), 171
auto keyword, 337
auto-typed variables, C++, 335-337
average(), 75
avoiding memory leaks, 150-151

B

bad luck, number 13, 108
Badger program, 85
Badger.cpp, 85
BadTeacher.cpp, 91
base 2, 412
base 10, 410-412
base class method, hiding, 247-249
base constructors, passing arguments to, 241-245
base methods, calling, 249-250
Big Ten, 387
binary numbers, 409, 413-414
  base 2, 412
bits, 413
black boxes, color of, 93
Box.cpp, 102
BoxMaker.cpp, 89
braces, code, 401-402
breaking out of loops, 83-84
BridgeKeeper.cpp, 105
bugs, 389-390
bulletproof programs, 390
Bushnel, R. G., 285
bytes, 413

C

C++, 5-6, 331
  auto-typed variables, 335-337
  compile-time constant expressions, 333-335
  errors, 333
  for loop, 338
  null pointer constants, 332-333
  reasons for using, 15-16
    styles of programming, 16-18
  C++-style comment, 22. See also double slash (//) comment
  C++0X, 331, 338
  C-style comments, 22. See also slash-star (/*) comments
Calculator, 25
Calculator.cpp, 24
  calling base methods, 249-250
camel case, 404
capitalization in code, 404
carboxylic acid with potassium ions, 369
case-sensitivity, variable names, 35
catch statements, 396
catching exceptions, 395-396
  by reference and polymorphism, 397-400
cought exceptions, 391
character arrays, 104-106
Circle.cpp, 334-335
class constructors,
  writing, 211
class declarations,
  organizing, 126
class definitions in code, 406
class keyword, 113
class members,
  accessing, 114
class polymorphism, 19
Classe, A., 285
classes, 112, 134
  adding simulating alarm systems, 348
  containment of, 307-312
  accessing
    members, 312
  copying by value versus reference, 313
  filtering members, 312-313
declaring, 113
designing
  PostMaster, 353-354
  simulating alarm systems, 347-348
friends, 313-314
objects, defining, 114
  with other classes as member data, 129, 132-133
private versus protected, 236-238
clown collete, 197
code
  program text
    guidelines, 403
  spelling/capitalization, 404
  writing professional-quality code, 400-401
    access, 405
    braces, 401-402
    class definitions, 406
    comments, 404-405
    const, 406
    identifier names, 403-404
    include files, 406
    long lines, 402
    program text, 403
    spelling and capitalization of names, 404
    switch statements, 402

How can we make this index more useful? Email us at indexes@samspublishing.com
code listings

driver programs, 367
polymorphic
exceptions, 400
Combat.cpp, 336-337
combining operators, 47
comments, 22-23
C++-style, 22
C-style, 22
double slash (//), 22, 26
in code, 404-405
slash-star (/*), 22, 26
star-slash (*), 22, 26
compile-time constant
expressions, 333-335
compiler and linker, 6
compilers, 16
constructors, 120-122
finding, 6-9
compiling, 16
source code, 9-10
components of linked lists, 291-299
compound if statements, 54-56
compound statements, 44
computer memory, 137
conceptualization, simulating
alarm systems, 345

const
in code, 406
member functions,
125-126
reasons for using, 165
const member functions,
163-164
const pointers, 162-164
passing, 188-190
constant expressions, 334
Constantinople, 251
constants, 37-38
defining, 38
enumerated constants,
38-39
symbolic, 40
constexpr keyword, 335
ConstPasser.cpp, 188-190
ConstPointer.cpp, 163
constructors, 119
copy constructors, 206-210
default constructors, 119
inheritance, 238-240
passing arguments to,
241-245
provided by compilers,
120-122
virtual copy constructors,
261-264

containment of classes,
307-312
accessing members, 312
copying by value versus
reference, 313
filtering members,
312-313
continue statement, 84-85
continuing to the next loop,
84-85
conversion operators,
225-227
int(), 227-228
convert() function, 67-69
copies, shallow copies, 206
copy constructor, 229
copy constructors, 206-210
copying
by value versus reference,
contained class, 313
strings, 106-107
cost of virtual member func-
tions, 264
Coulier, Dave, 230
Counter.cpp, 216
Counter2.cpp, 217
Counter3.cpp, 219-220
Counter4.cpp, 221
Counter5.cpp, 225
Counter6.cpp, 226
Counter7.cpp, 227-228
Dangling pointers, 161-162
Data, manipulating with pointers, 143-144
Data members, 113
  accessing with pointers, 157-158
Data types, 30
  abstract, 273-276
DataMember.cpp, 158
Decimal math, 409
Declaring
  classes, 113
  functions, 64-65
  on the heap, 165
decrement operators, 47
DeepCopy.cpp, 207-209
default constructors, 119
default function parameters, 72-74
default values, 203-204
Defining
  functions, 65-66
  objects, 114
delegation of responsibilities, linked lists, 291
delete keyword, 148-150
Deleting objects, 118-119
  from heap, 155-157
dereference operator (*), 142
dereferencing, 317
derivation
  inheritance and, 234
  syntax of, 235-236
designing
  classes
    PostMaster, 353-354
    simulating alarm systems, 347-348
  interfaces, PostMaster, 356-358
  PostMaster, ongoing design considerations, 361-362
  PostMasterMessage class, 359
destructors, 119
  inheritance, 238-240
  virtual destructors, 261
development cycle, 343-344
Diogenes, 301
Disney World, 183
dividing up projects,
    PostMaster, 352-353
do-while loops, 85-86
dot operator (.), 114
double slash (//) comments, 22, 26
doubly linked lists, 290
drawShape() function, 202
driver programs, PostMaster, 362-368
Driver.cpp, 362-368
Dusky Seaside Sparrow, 183

E
editing Path, G++, 8
Elements, arrays, 97
else keyword, 53-54
Employee.cpp, 310-312
Encapsulation, 112, 347
  object-oriented programming, 18
enumerated constants, 38-39
equality operator (==), 225
errors, 390
  arrays, 99
  C++, 333
event loops, simulating alarm systems, 348-350
exception handling, 390-391
Exception.cpp, 392-395
Exceptions, 148, 390-391
  how they are used, 391-395
  polymorphic, 400
  try blocks, 395
    catching exceptions, 395-396
    catching exceptions by reference and polymorphism, 397-400
Expression.cpp

Expression.cpp, 45
expressions, 44-45
values, 58

F
factor(), 180
Favre, Henri, 369
Fell, Norman, 40
fence post error, 99
Fifteens program, 84
Fifteens.cpp, 84
filtering members of contained class, 312-313
findArea() function, 65-66, 74
finding compilers, 6-9
for loops, 86-88
advanced, 88
C++, 338
Fourteens program, 83
Fourteens.cpp, 83
friend classes, 313-314
friend functions, 314
arrays of pointers, 317-319
arrays of pointers to member functions, 325-326
passing pointers to, 319-322
pointers to, 314-317
friends, 314
arrays of pointers, 317-319
arrays of pointers to member functions, 325-326
passing pointers to functions, 319-322
pointers, 314-317
pointers to member functions, 322-325
typedef with pointers to functions, 322
function arguments, 26
passing by reference, 174-176
function definitions, organizing, 126
function headers, 179
function names, 24
function overloading, 74-75
inline functions, 75-76
function parameters, 26
function pointers, 314-317
function polymorphism, 74
function prototypes, 179
functional polymorphism, 19
FunctionOne(), 188
FunctionPasser.cpp, 319-322
FunctionPointer.cpp, 315-317
functions, 21-24, 63
add(), 25
arguments, 24-25
average(), 75
convert(), 67-69
declaring, 64-65
defining, 65-66
drawShape(), 202
factor(), 180
findArea(), 65-66, 74
getSpeed(), 125, 134
isLeapYear(), 72
main(), 25, 63
operator++(), 219
overriding inheritance, 245-247
parameters, 69-70
default, 72-74
returning values from, 70-72
setSpeed(), 125
signature, 245
sizeof(), 30
someFunc(), 265
store(), 74
stubbing out, 235
swap(), 70, 176-177
TheFunction(), 194
variables
global variables, 68-69
local variables, 66-68
virtual functions, 276-277
implementing, 277-280
virtual member functions,
how they work, 257-258
FunctionTwo(), 188

G

G++, 7
editing Path, 8
getSpeed() definition, 127
global variables, 76
functions, 68-69
Hexadecimal numbers, 413
global variables, 76
functions, 68-69
Global.cpp, 68
hexadecimal numbers, 413
Gnomes, 123
gnoming, 123
gold, panning for in U.S., 387
Grader.cpp, 53
Granholm, Jackson W., 27
grasshopper ice cream, 134
Griebel, Phillip, 123

H

handling the unexpected,
390-391
Hayes, Woody, 327
head nodes, 289
head, 146-148, 152
avoiding memory leaks,
150-151
declarations on, 165
delete keyword, 148-150
member data, 158-159
new keyword, 148
objects
creating, 155
deleting, 155-157
returning references to
objects, 194-196
Heap.cpp, 149
HeapAccessor.cpp, 157
HeapCreator.cpp, 156
hexadecimal numbers, 409,
414-417
hiding base class method,
247-249
hierarchies
abstract data types,
280-284
rooted versus nonrooted,
354-356
high-level design, simulating
alarm systems, 346
horseradish hot, 211

I

IDE (integrated development
environments), 12
identifier names, code,
403-404
identifiers, naming
guidelines, 403
if statements, compound,
54-56
If-Else conditional
statements, 53
compound if statements,
54-56
else clause, 53-54
implementation
inline, 127-129
member functions,
116-118
polymorphism with virtual
methods, 253-257
swap() with references,
177-178
virtual functions, abstract
data types, 277-280
include files, 20, 406
increment methods, writing
for operator overloading,
217-218
increment operators, 47
indirection operators, 142
infinite loops, 82

How can we make this index more useful? Email us at indexes@samspublishing.com
inheritance, 233
animals and, 234
constructors, 238-240
derivation and, 234-236
destructors, 238-240
object-oriented programming, 18
overriding functions, 245-247
potential problems, 270
single inheritance, problems with, 269-273
initializing
arrays, 100
multidimensional arrays, 102-103
objects, 205
initializing variables, 35
inline functions, 75-76
inline implementation, 127-129
inline keyword, 127
instance variables, 113
instances, templates, 374
instantiation, objects, 374
int() operator, 227-228
integrated development environments (IDE), 12
interfaces, designing for PostMaster, 356-358
International Code of Zoological Nomenclature, 59
interpreters, 16
isLeapYear() function, 72
Istanbul, 251
iteration, 81

J
jostream files, 20

K
KB, 413
keywords
auto, 337
class, 113
constexpr, 335
delete, 148-150
inline, 127
new, 148
kings, 152
kludge, 26
Klugman, Jack, 40

L
late binding, 257
lawn gnomes, 123
Leak.cpp, 194
LeapYear program, 70-71
LeapYear.cpp, 70
limitations on operator overloading, 222
linked lists, 289
allocating, 296
as objects, 299-300
components, 291-299
delegation of responsibilities, 291
infinite loops, 296
singly linked, 290
LinkedList.cpp, 291-295
linking source code, 9-10
listings
Animal.cpp, 280-283
Area.cpp, 65, 73
ArrayFunction.cpp, 317-319
Assignment.cpp, 172, 224-225
Badger.cpp, 85
BadTeacher.cpp, 91
Box.cpp, 102
BoxMaker.cpp, 89
BridgeKeeper.cpp, 105
local variables, functions, 66-68

logical operators
AND, 56
NOT, 56-57
OR, 56-57
relational precedence, 57-58

long lines of code, 402
looping, 81
loops
breaking out of, 83-84
continuing to the next, 84-85
do-while loops, 85-86
for loops, 86-88
advanced, 88
C++, 338
nested loops, 89-90
while loops, 81-83

low-level design, simulating alarm systems, 346

M

machine language, 15
main() function, 23-25, 63
Mammal1.cpp, 235
Mammal2.cpp, 237-238
Mammal3.cpp, 239-240
Mammal4.cpp, 241-243
Mammal5.cpp, 246
Mammal6.cpp, 247
Mammal7.cpp, 249-250
Mammal8.cpp, 254-255
Mammal9.cpp, 255-257
Mammal10.cpp, 259-260
Mammal11.cpp, 261-263
Mammal12.cpp, 269-270
Mammal13.cpp, 271-272

manipulating data with pointers, 143-144

mathematical operators, 46

member data
classes as, 129, 132-133
heap, 158-159

member functions
arrays of pointers to,
325-326
const, 125-126, 163-164
implementing, 116-118
overloaded, 201-203
pointers to, 322-325

member variables, 113-114
private versus public access, 115

MemberPointer.cpp, 322-325
members, 112
memory, 137
heap, 146-148

avoiding memory leaks, 150-151

methods
base class method, hiding, 247-249
base method, calling, 249-250
virtual methods, implementing polymorphism, 253-257

MinGW, 7
mistakes, 390
modulus operator, 46
Morissette, Alanis, 229
Motto.cpp, 10, 19
source code, 21-22

MPFunction.cpp, 325-326
multidimensional arrays, 101-102
    initializing, 102-103
memory, 103
MultTable program, 87
MultTable.cpp, 87

N
naming animals, 59
Nawrocki, Tom, 40
negative numbers, 59
nested loops, 89-90
new keyword, 148
NewGrader program, 55
NewGrader.cpp, 54
NewRectangle.cpp, 36
NewTricycle.cpp, 120
No. 1 pencils, 77
nodes, 289
    inserting, 297
non-pointers, 141
non-rooted hierarchies versus
    rooted hierarchies, 354-356
NOT operator, 56-57
NULL, 159
null pointer constants,
    332-333
null pointers, 148, 174
null references, 174
numbers
    base 10, 410-412
    base 2, 412
    binary numbers, 412-414
        base 2, 412
    bits, 413
    bytes, 413
    gigabytes, 413
    hexadecimal numbers,
        414-417
    KB, 413
    nybbles, 413
nybbles, 413
initializing, 205
linked lists as, 299-300
on the heap, returning
    references to, 194-196
references, 174
returning references to,
    193-194
Ohio College of Clowning
    Arts, 197
OOP (object-oriented pro-
    gramming), 18, 122
    encapsulation, 18
    inheritance and reuse, 18
    polymorphism, 19
operator overloading,
    215-217
    addition operator,
        220-222
    limitations, 222
    operator=(), 223-225
    postfix operator, 219-220
    writing increment
        methods, 217-218
operator++() member
    function, 219
operator=(), 223-225
operators, 45
    addition operator, over-
        loading, 220-222
    assignment operators, 46
    combining, 47

How can we make this index more useful? Email us at indexes@samspublishing.com
operators

conversion operators, 225-227
  int(), 227-228
decrement, 47
dynamic cast, 273
equality (==), 225
increment, 47
indirection operator, 142
indirection (*), 142
logical operators
  AND, 56
  NOT, 56-57
  OR, 56-57
  relational precedence, 57-58
mathematical operators, 46
pointers, 142
postfix, 48-49
  overloading, 219-220
  precedence, 49-51
  prefix, 48-49
  relational, 51-52
OR operator, 56-57
organizing class declarations and function definitions, 126
overloaded member functions, 201-203
  default values, 210
overloading
  functions, 74-75
    inline functions, 75-76
  operators. See operator overloading
  versus overriding, 247
overriding
  functions, inheritance, 245-247
  versus overloading, 247

P-Q

parameterized List objects,
declaring (templates), 374, 381
parameters, functions, 69-70
  default, 72-74
ParamList.cpp, 375-381
parentheses (), 51, 59, 157
passing
  arguments to base constructors, 241-245
  const pointers, 188-190
  function arguments by reference, 174-176
  passing by references, 185-188
  passing pointers to functions, 319-322
Path, editing, 8
peaks array, 97
pencils, No. 1, 77
photography, red eyes, 165
Pointer.cpp, 143
PointerCheck.cpp, 144
pointers, 137-140, 143
  accessing data members, 157-158
  addresses, 144-146
  const, 162-164
    passing, 188-190
  dangling pointers, 161-162
to functions, 314-317
  arrays, 317-319
  indirection operators, 142
  manipulating data, 143-144
to member functions, 322-325
  arrays, 325-326
  null, 148, 174
  owner of, 196
  passing to functions, 319-322
  reasons for using, 146
  references as alternatives, 191-192
  storing addresses, 140-142
storing memory addresses, 161
stray, 161-162
swap(), 176-177
this, 160-161
typedef with pointers to functions, 322
when to use, 192
wild, 141
PointerSwap.cpp, 176
PolyException.cpp, 397-400
polymorphism
catching exceptions, 397-400
implementing with virtual methods, 253-257
object-oriented programming, 19
virtual methods, 257
postfix operators, 48-49
overloading, 219-220
PostMaster, 351
80/80 rule, 358-359
APIs, 359-361
design considerations, 361-362
designing interfaces, 356-358
PostMasterMessage class, 359
dividing projects up, 352-353
driver programs, 362-368
initial class design, 353-354
message format, 353
programming in large groups, 361
prototypes, building, 358
rooted hierarchies versus non-rooted, 354-356
PostMaster Professional, 351
PostMasterMessage class, designing, 359
PostMasterMessage.cpp, 360-361
precedence, operators, 49-51
prefix operators, 48-49
preprocessor directives, 20-21
private versus protected, 236-238
private access versus public access, 115-116
problem space, 346
problems with single inheritance, 269-273
professional-quality code, writing
access, 405
braces, 401-402
class definitions, 406
comments, 404-405
const, 406
identifier names, 403-404
include files, 406
long lines, 402
program text, 403
switch statements, 404
program text, code, 403
programming large groups, PostMaster, 361
programming languages, machine language, 15
programming styles, 16-18
programs
build/buy question, 356
creating, 10-11
high-level, 346
protected versus private, 236-238
prototypes, building for PostMaster, 358
public access versus private access, 115-116

R

Rectangle.cpp, 35, 130, 201-202
Rectangle.hpp, 129
Rectangle2.cpp, 203-204
red eyes in photographs, 165
Reference.cpp

Reference.cpp, 170
Reference2.cpp, 171
references, 169
  address of operators, 171-173
  as alternatives to pointers, 191-192
  catching exceptions, 397-400
  creating, 170-171
  determining what can be referenced, 173-174
  implementing swap(), 177-178
null references, 174
passing by, 185-188
passing function arguments, 174-176
returning multiple values, 179-181
returning to objects, 193-194
returning to objects on the heap, 194-196
returning values, 181-182
when to use, 192
ReferenceSwap.cpp, 177
RefPasser.cpp, 191-192
relational operators, 51-52
relational precedence, 57-58
requirements, simulating alarm systems, 345
responsibilities, delegation of (linked lists), 291
return statements, 24
returning
  references to objects, 193-194
  references to objects on the heap, 194-196
  values by reference, 181-182
returning values from functions, 70-72
ReturnPointer program, 181
ReturnPointer.cpp, 179
ReturnRef.cpp, 193-194
ReturnReference.cpp, 181
returnType, 217
reuse, object-oriented programming, 18
rooted hierarchies versus non-rooted hierarchies, 354-356
Rudolph the Red-Nosed Reindeer, 265
runtime binding, 257
S
semicolon, 44
setSpeed() function, 125
shallow copies, 206
Shape.cpp, 273-275
Shape2.cpp, 277-280
signatures, 24
  functions, 245
signed short integers, 33
signed variables, 32
Silbo Gomero, 285
SimpleEvent.cpp, 348-350
simulations, 344
simulating alarm systems, 344
  adding more classes, 348
  analysis and requirements, 345
  conceptualization, 345
  designing classes, 347-348
  event loops, 348-350
  high-level and low-level design, 346
single inheritance, problems with, 269-273
singly linked lists, 290
sizeof(), 30
Sizer.cpp, 31
slash-star (/*) comments, 22, 26
slicing virtual member functions, 259-261
solution space, 346
someFunc(), 265
source code
  compiling and linking, 9-10
  Motto.cpp, 21-22
spelling in code, 404
stack, 146-148
  avoiding memory leaks, 150-151
  delete keyword, 148-150
  new keyword, 148
star-slash (*) comments, 22, 26
statements, 43
  compound statements, 44
  continue, 84-85
  if-else, 53
    compound if statements, 54-56
    else clause, 53-54
    switch, 90-92
  whitespace, 43-44
static member data, 303-305
static member functions, 305-307
  StaticCat.cpp, 304
  StaticFunction.cpp, 305-306
store() function, 74
storing
  addresses in pointers, 140-142
  variables in memory, 30-31
stray pointers, 161-162
strcpy(), 106
String.hpp, 307-310
StringCopier.cpp, 106
strings, copying, 106-107
strncpy(), 106
Stroustrup, Bjarne, 5, 338
Strupper, Everett, 407
stubbing out, 235
styles of programming, 16-18
swap() function, 70
  implementing with references, 177-178
  swap() pointers, 176-177
Swapper.cpp, 332-333
switch statements, 90-92, 402
switch-case conditional, 93
symbolic constants, 40
T
tail nodes, 289
Temperature.cpp, 66
template items, 381-386
TemplateList.cpp, 381-386
templates, 373-374
  defined, 374-381
  instances, 374
  passing, 386
text editors, 12
TheFunction(), 194
Thirteens program, 82
Thirteens.cpp, 82
this pointer, 160-161
  creating, 161
This.cpp, 160
thrown exceptions, 391
Tricycle.cpp, 116-117, 128
Tricycle.hpp, 127
try blocks, 392, 395
  catching exceptions, 395-396
  by reference and polymorphism, 397-400
  throwing, 391
type definitions, 36-37
typedef, 36-37
  pointers to functions, 322
types, 111
  creating new, 112

U
UML (Unified Modeling Language), 345
unsigned short integers, 33
unsigned variables, 32
use cases, 345

How can we make this index more useful? Email us at indexes@samspublishing.com
v-pointers

V

v-pointers, 258
ValuePasser.cpp, 175
values
   assigning to variables, 35-36
default values, 203-204
expressions, 58
returning by reference, 181-182
returning from functions, 70-72
returning multiple values, references, 179-181
variable types, 32-33
variables, 29, 143
   adding to C++, 49
   assigning values to, 35-36
   auto-typed variables, 335-337
classes, 113
defining, 33-35
global, 76
   functions, 68-69
local variables, functions, 66-68
signed, 32
storing in memory, 30-31
unsigned, 32
variable types, 32-33
virtual copy constructors, virtual member functions, 261-264
virtual destructors, virtual member functions, 261
virtual functions, 276-277
   implementing, 277-280
virtual member functions, 258
   cost of, 264
   how they work, 257-258
   slicing, 259-261
virtual copy constructors, 261-264
   virtual destructors, 261
virtual methods, implementing polymorphism, 253-257
virtual table pointers, 258
vptr (virtual table pointers), 258

W-X

writing
   class constructors, 211
   increment methods, operator overloading, 217-218
   past the end of arrays, 99
   professional-quality code, 400-401
     access, 405
     braces, 401-402
     class definitions, 406
     comments, 404-405
     const, 406
     identifier names, 403-404
     include files, 406
     long lines, 402
     program text, 403
     spelling and capitalization, 404
     switch statements, 402

Y-Z

Years.cpp, 48
“You Oughta Know,” 229