COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

INTRODUCTION TO COMPUTING

Unit 11: Introduction to Programming

Introduction

Computer programming is the process of designing and writing instructions that a computer can execute to perform specific tasks. These instructions, known as code, are written using programming languages that bridge the gap between human logic and machine operations. A programming language is a formal language used to communicate instructions to a computer. It provides a structured way for programmers to write code that tells the computer what to do. Programming languages consist of a set of rules (syntax) and keywords that define how programs are written and interpreted. They allow humans to create software, control hardware, and solve problems by expressing logic and computations in a form that machines can process. Examples of programming languages include Python, Java, C++, and JavaScript. In this section of CS 110, you will be introduced to the fundamental principles of programming in C++, including syntax, control structures, data types, and problem-solving techniques.

A Brief History of C++

Computer languages have undergone dramatic evolution since the first electronic computers were built to assist in telemetry calculations during World War II. Early on, programmers worked with the most primitive computer instructions: machine language. These instructions were represented by long strings of ones and zeroes. Soon, assemblers were invented to map machine instructions to humanreadable and -manageable mnemonics, such as `ADD` and `MOV`.

In time, higher-level languages evolved, such as BASIC and COBOL. These languages let people work with something approximating words and sentences, such as `Let I = 100`. These instructions were translated back into machine language by interpreters and compilers. An interpreter translates a program as it reads it, turning the program instructions, or code, directly into actions. A compiler translates the code into an intermediary form. This step is called compiling, and produces an object file. The compiler then invokes a linker, which turns the object file into an executable program.

Because interpreters read the code as it is written and execute the code on the spot, interpreters are easy for the programmer to work with. Compilers, however, introduce the extra steps of compiling and linking the code, which is inconvenient. Compilers produce a program that is very fast each time it is run. However, the time-consuming task of translating the source code into machine language has already been accomplished.

Another advantage of many compiled languages like C++ is that you can distribute the executable program to people who don't have the compiler. With an interpretive language, you must have the language to run the program.

For many years, the principle goal of computer programmers was to write short pieces of code that would execute quickly. The program needed to be small, because memory was expensive, and it needed to be fast, because processing power was also expensive. As computers have become smaller, cheaper, and faster, and as the cost of memory has fallen, these priorities have changed. Today the cost of a programmer's time far outweighs the cost of most of the computers in use by businesses. Well-written, easy-to-maintain code is at a premium. Easy- to-maintain means that as business requirements change, the program can be extended and enhanced without great expense.

Programs

The word program is used in two ways: to describe individual instructions, or source code, created by the programmer, and to describe an entire piece of executable software. This distinction can cause enormous confusion, so we will try to distinguish between the source code on one hand, and the executable on the other.

New Term: A *program* can be defined as either a set of written instructions created by a programmer or an executable piece of software.

Source code can be turned into an executable program in two ways: Interpreters translate the source code into computer instructions, and the computer acts on those instructions immediately.
Alternatively, compilers translate source code into a program, which you can run at a later time. While interpreters are easier to work with, most serious programming is done with compilers because compiled code runs much faster. C++ is a compiled language.

Solving Problems

The problems programmers are asked to solve have been changing. Twenty years ago, programs were created to manage large amounts of raw data. The people writing the code and the people using the program were all computer professionals. Today, computers are in use by far more people, and most know very little about how computers and programs work. Computers are tools used by people who are more interested in solving their business problems than struggling with the computer.

Ironically, in order to become easier to use for this new audience, programs have become far more sophisticated. Gone are the days when users typed in cryptic commands at esoteric prompts, only to see a stream of raw data. Today's programs use sophisticated "user-friendly interfaces," involving multiple windows, menus, dialog boxes, and the myriad of metaphors with which we've all become familiar. The programs written to support this new approach are far more complex than those written just ten years ago.

Preparing to Program

C++, perhaps more than other languages, demands that the programmer design the program before writing it. Trivial problems, such as the ones discussed in the first few chapters of this book, don't require much design. Complex problems, however, such as the ones professional programmers are challenged with every day, do require design, and the more thorough the design, the more likely it is that the program will solve

the problems it is designed to solve, on time and on budget. A good design also makes for a program that is relatively bug-free and easy to maintain. It has been estimated that fully 90 percent of the cost of software is the combined cost of debugging and maintenance. To the extent that good design can reduce those costs, it can have a significant impact on the bottom-line cost of the project.

The first question you need to ask when preparing to design any program is, "What is the problem I'm trying to solve?" Every program should have a clear, well-articulated goal, and you'll find that even the simplest programs in this book do so.

The second question every good programmer asks is, "Can this be accomplished without resorting to writing custom software?" Reusing an old program, using pen and paper, or buying software off the shelf is often a better solution to a problem than writing something new. The programmer who can offer these alternatives will never suffer from lack of work; finding less-expensive solutions to today's problems will always generate new opportunities later.

Assuming you understand the problem, and it requires writing a new program, you are ready to begin your design.

Your Development Environment

Programming is mostly done using an Integrated Development Environment (IDE), which, although not strictly required, significantly simplifies the tasks of writing, debugging, and executing code. An Integrated Development Environment (IDE) is a software application that provides a comprehensive interface for writing, compiling, and executing programs. It combines several essential programming tools into a single platform, including a code editor, compiler or interpreter, debugger, and often a graphical user interface (GUI) builder. By integrating these tools, an IDE simplifies the programming process and enhances productivity. Programmers can write and test their code in the same environment, which reduces the complexity of switching between different applications. Overall, an IDE makes it easier to develop, compile, and run programs efficiently, especially for beginners learning to code.

The files you create with your editor are called source files, and for C++ they typically are named with the extension `.CPP`, `.CP`, or `.C`. In this course, we'll name all the source code files with the `.CPP` extension, but check your compiler for what it needs.

NOTE: Please download a suitable Integrated Development Environment (IDE) that is compatible with your computer, smartphone, or tablet to begin practicing programming. For smartphones, a suitable CPP compiler can be found on the app store. For Desktop/ laptops, you can download suitable IDE online. Examples of available IDEs for computers include Code::blocks, Visual Studio Code, NetBeans, Dev-C++, Visual Studio, Lazarus, etc.

Compiling the Source Code

Although the source code in your file is somewhat cryptic, and anyone who doesn't know C++ will struggle to understand what it is for, it is still in what we call human-readable form. Your source code file is not a program, and it can't be executed, or run, as a program can. To turn your source code into a program, you use a compiler.

The Development Cycle

If every program worked the first time you tried it, that would be the complete development cycle: Write the program, compile the source code, link the program, and run it. Unfortunately, almost every program, no matter how trivial, can and will have errors, or bugs, in the program. Some bugs will cause the compile to fail, some will cause the link to fail, and some will only show up when you run the program.

Whatever type of bug you find, you must fix it, and that involves editing your source code, recompiling and relinking, and then rerunning the program.

HELLO.CPP Your First C++ Program

Traditional programming books begin by writing the words `Hello World` to the screen, or a variation on that statement. This time-honored tradition is carried on here.

Type the first program directly into your editor, exactly as shown. Once you are certain it is correct, save the file, compile it, link it, and run it. It will print the words `Hello World` to your screen.

Don't worry too much about how it works, this is really just to get you comfortable with the development cycle. Every aspect of this program will be covered over the next couple of days.

WARNING: The following listing contains line numbers on the left. These numbers are for reference within the notes. They should not be typed in to your editor. For example, in line 1 of Listing 1.1, you should enter:

```
#include <iostream.h>
```

Listing 1.1. HELLO.CPP, the Hello World program.

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:    cout << "Hello World!\n";
6:        return 0;
7: }
```

Make certain you enter this exactly as shown. Pay careful attention to the punctuation. The << in line 5 is the redirection symbol, produced on most keyboards by holding the Shift key and pressing the comma key twice. Line 5 ends with a semicolon; don't leave this off!

Also check to make sure you are following your compiler directions properly. Most compilers will link automatically, but check your documentation. If you get errors, look over your code carefully and determine how it is different from the above. If you see an error on line 1, such as `cannot find file iostream.h`, check your compiler documentation for directions on setting up your `include` path or environment variables. If you receive an error that there is no prototype for `main`, add the line `int main();` just before line 3. You will need to add this line before the beginning of the `main` function in every program in this book. Most compilers don't require this, but a few do.

Your finished program will look like this:

```
1: #include <iostream.h>
2:
3:
4: int main();
5: {
6: cout <<"Hello World!\n";
7:     return 0;
8: }
```

Try running the program. Congratulations you've just entered, compiled, and run your first C++ program. It may not look like much, but almost every professional C++ programmer started out with this exact program.

Compile Errors

Compile-time errors can occur for any number of reasons. Usually they are a result of a typo or other inadvertent minor error. Good compilers will not only tell you what you did wrong, they'll point you to the exact place in your code where you made the mistake. The great ones will even suggest a remedy!

You can see this by intentionally putting an error into your program. If `HELLO.CPP` ran smoothly, edit it now and remove the closing brace on line 6. Your program will now look like Listing 1.2.

Listing 1.2. Demonstration of compiler error.

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:   cout << "Hello World!\n";
6: return 0;
```

Recompile your program and you should see an error that looks similar to the following:

```
Hello.cpp, line 5: Compound statement missing terminating } in function
main().
```

This error tells you the file and line number of the problem, and what the problem is (although I admit it is somewhat cryptic). Note that the error message points you to line 5. The compiler wasn't sure if you intended to put the closing brace before or after the `cout` statement on line 5. Sometimes the errors just get you to the general vicinity of the problem. If a compiler could perfectly identify every problem, it would fix the code itself.

Exercises

1. Look at the following program and try to guess what it does without running it.

```
#include <iostream.h>

int main()
{
      int x = 5;
      int y = 7;
      cout "\n";
      cout << x + y << " " << x * y;
      cout "\n";
      return 0;
}
```

2. Type in the program from Exercise 1, and then compile and link it. What does it do? Does it do what you guessed?

3. Type in the following program and compile it. What error do you receive?

```
include <iostream.h>

int main()
{
      cout << "Hello World\n";
      return 0;
}
```

4. Fix the error in the program in Exercise 3, and recompile and run it. What does it do?

COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

INTRODUCTION TO COMPUTING

Unit 12: The Parts of a C++ Program

Introduction

C++ programs consist of objects, functions, variables, and other component parts. To get a sense of how a program fits together, you must see a complete working program. Today you learn

- The parts of a C++ program.
- How the parts work together.
- What a function is and what it does.

A Simple Program

Even the simple program HELLO.CPP from Day 1, "Getting Started," had many interesting parts. This section will review this program in more detail. Listing 2.1 reproduces the original version of HELLO.CPP for your convenience.

Listing 2.1. HELLO.CPP demonstrates the parts of a C++ program.

```
1: #include <iostream.h>
2:
3:  int main()
4: {
5:    cout << "Hello World!\n";
6:      return 0;
7: }
Hello World!
```

On line 1, the file iostream.h is included in the file. The first character is the # symbol, which is a signal to the preprocessor. Each time you start your compiler, the preprocessor is run. The preprocessor reads through your source code, looking for lines that begin with the pound symbol (#), and acts on those lines before the compiler runs.

include is a preprocessor instruction that says, "What follows is a filename. Find that file and read it in right here." The angle brackets around the filename tell the preprocessor to look in all the usual places for this file. If your compiler is set up correctly, the angle brackets will cause the preprocessor to look for the file iostream.h in the directory that holds all the H files for your compiler. The file iostream.h

(Input-Output-Stream) is used by cout, which assists with writing to the screen. The effect of line 1 is to include the file iostream.h into this program as if you had typed it in yourself.

New Term: The preprocessor runs before your compiler each time the compiler is invoked. The preprocessor translates any line that begins with a pound symbol (#) into a special command, getting your code file ready for the compiler.

Line 3 begins the actual program with a function named main(). Every C++ program has a main() function. In general, a function is a block of code that performs one or more actions. Usually functions are invoked or called by other functions, but main() is special. When your program starts, main() is called automatically.

main(), like all functions, must state what kind of value it will return. The return value type for main() in HELLO.CPP is void, which means that this function will not return any value at all. Returning values from functions is discussed in detail on Day 4, "Expressions and Statements."

All functions begin with an opening brace ({) and end with a closing brace (}). The braces for the main() function are on lines 4 and 7. Everything between the opening and closing braces is considered a part of the function.

The meat and potatoes of this program is on line 5. The object cout is used to print a message to the screen. We'll cover objects in general on Day 6, "Basic Classes," and cout and its related object cin in detail on Day 17, "The Preprocessor." These two objects, cout and cin, are used in C++ to print strings and values to the screen. A string is just a set of characters.

Here's how cout is used: type the word cout, followed by the output redirection operator (<<). Whatever follows the output redirection operator is written to the screen. If you want a string of characters written, be sure to enclose them in double quotes ("), as shown on line 5.

New Term: A text string is a series of printable characters.

The final two characters, \n, tell cout to put a new line after the words Hello World!

All ANSI-compliant programs declare main() to return an int. This value is "returned" to the operating system when your program completes. Some programmers signal an error by returning the value 1. In this book, main() will always return 0.

The main() function ends on line 7 with the closing brace.

A Brief Look at cout

To print a value to the screen, write the word cout, followed by the insertion operator (<<), which you create by typing the less-than character (<) twice. Even though this is two characters, C++ treats it as one.

Follow the insertion character with your data. Listing 2.2 illustrates how this is used. Type in the example exactly as written, except substitute your own name where you see John Doe (unless your name is John Doe, in which case leave it just the way it is; it's perfect-- but I'm still not splitting royalties!).

Listing 2.2. (a) Using cout.

```
1:      // Listing 2.2 using cout
2:
3:      #include <iostream.h>
4:      int main()
5:      {
6:         cout << "Hello there.\n";
7:         cout << "Here is 5: " << 5 << "\n";
8:         cout << "The manipulator endl writes a new line to the
screen." <<
                      endl;
9:         cout << "Here is a very big number:\t" << 70000 << endl;
10:        cout << "Here is the sum of 8 and 5:\t" << 8+5 << endl;
11:        cout << "Here's a fraction:\t\t" << (float) 5/8 << endl;
12:        cout << "And a very very big number:\t" << (double) 7000
* 7000 <<
                      endl;
13:        cout << "Don't forget to replace John Doe with your
name...\n";
14:        cout << "John Doe is a C++ programmer!\n";
15:         return 0;
16: }
```

Listing 2.2.(b) Output when the program is executed

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:      70000
Here is the sum of 8 and 5:     13
Here's a fraction:              0.625
And a very very big number:     4.9e+07 Don't forget to
replace John Doe with your name... John Doe is a C++
programmer!
```

On line 3, the statement #include <iostream.h> causes the iostream.h file to be added to your source code. This is required if you use cout and its related functions.

On line 6 is the simplest use of cout, printing a string or series of characters. The symbol \n is a special formatting character. It tells cout to print a newline character to the screen.

Three values are passed to cout on line 7, and each value is separated by the insertion operator. The first value is the string "Here is 5: ". Note the space after the colon. The space is part of the string. Next, the

value 5 is passed to the insertion operator and the newline character (always in double quotes or single quotes). This causes the line

```
Here is 5: 5
```

to be printed to the screen. Because there is no newline character after the first string, the next value is printed immediately afterwards. This is called concatenating the two values.

On line 8, an informative message is printed, and then the manipulator endl is used. The purpose of endl is to write a new line to the screen. (Other uses for endl are discussed on Day 16.)

On line 9, a new formatting character, \t, is introduced. This inserts a tab character and is used on lines 8-12 to line up the output. Line 9 shows that not only integers, but long integers as well can be printed. Line 10 demonstrates that cout will do simple addition. The value of 8+5 is passed to cout, but 13 is printed.

On line 11, the value 5/8 is inserted into cout. The term (float) tells cout that you want this value evaluated as a decimal equivalent, and so a fraction is printed. On line 12 the value 7000 * 7000 is given to cout, and the term (double) is used to tell cout that you want this to be printed using scientific notation. All of this will be explained on Day 3, "Variables and Constants," when data types are discussed.

On line 14, you substituted your name, and the output confirmed that you are indeed a C++ programmer. It must be true, because the computer said so!

Comments

When you are writing a program, it is always clear and self-evident what you are trying to do. Funny thing, though--a month later, when you return to the program, it can be quite confusing and unclear. I'm not sure how that confusion creeps into your program, but it always does.

To fight the onset of confusion, and to help others understand your code, you'll want to use comments. Comments are simply text that is ignored by the compiler, but that may inform the reader of what you are doing at any particular point in your program.

Types of Comments

C++ comments come in two flavors: the double-slash (//) comment, and the slash-star (/*) comment. The double-slash comment, which will be referred to as a C++-style comment, tells the compiler to ignore everything that follows this comment, until the end of the line.

The slash-star comment mark tells the compiler to ignore everything that follows until it finds a starslash (*/) comment mark. These marks will be referred to as C-style comments. Every /* must be matched with a closing */.

As you might guess, C-style comments are used in the C language as well, but C++-style comments are not part of the official definition of C.

Many C++ programmers use the C++-style comment most of the time, and reserve C-style comments for blocking out large blocks of a program. You can include C++-style comments within a block "commented out" by C-style comments; everything, including the C++-style comments, is ignored between the C-style comment marks.

Using Comments

As a general rule, the overall program should have comments at the beginning, telling you what the program does. Each function should also have comments explaining what the function does and what values it returns. Finally, any statement in your program that is obscure or less than obvious should be commented as well.

Listing 2.3 demonstrates the use of comments, showing that they do not affect the processing of the program or its output.

Listing 2.3. HELP.CPP demonstrates comments.

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:  /* this is a comment
6:  and it extends until the closing
7:  star-slash comment mark */
8:    cout << "Hello World!\n";
9:    // this comment ends at the end of the line
10:   cout << "That comment ended!\n";
11:
12:  // double slash comments can be alone on a line
13: /* as can slash-star comments */
14:     return 0;
15: }
Hello World!
That comment ended!
```

The comments on lines 5 through 7 are completely ignored by the compiler, as are the comments on lines 9, 12, and 13. The comment on line 9 ended with the end of the line, however, while the comments on lines 5 and 13 required a closing comment mark.

Comments at the Top of Each File

It is a good idea to put a comment block at the top of every file you write. The exact style of this block of comments is a matter of individual taste, but every such header should include at least the following information:

- The name of the function or program.

- The name of the file.
- What the function or program does.
- A description of how the program works.
- The author's name.
- A revision history (notes on each change made).
- What compilers, linkers, and other tools were used to make the program.
- Additional notes as needed.

For example, the following block of comments might appear at the top of the Hello World program.

```
/*************************************************************

Program:       Hello World

File:          Hello.cpp

Function:      Main (complete program listing in this file) Description:

Prints the words "Hello world" to the screen

Author:        Jesse Liberty (jl)

Environment:   Turbo C++ version 4, 486/66 32mb RAM, Windows 3.1
DOS 6.0.  EasyWin module. Notes:         This is an introductory,
sample program.

Revisions:     1.00  10/1/94 (jl) First release
               1.01  10/2/94 (jl) Capitalized "World"

*************************************************************/
```

It is very important that you keep the notes and descriptions up-to-date. A common problem with headers like this is that they are neglected after their initial creation, and over time they become increasingly misleading. When properly maintained, however, they can be an invaluable guide to the overall program.

The listings in the rest of this book will leave off the headings in an attempt to save room. That does not diminish their importance, however, so they will appear in the programs provided at the end of each week.

A Final Word of Caution About Comments

Comments that state the obvious are less than useful. In fact, they can be counterproductive, because the code may change and the programmer may neglect to update the comment. What is obvious to one person may be obscure to another, however, so judgment is required.

The bottom line is that comments should not say what is happening, they should say why it is happening.

DO add comments to your code. **DO** keep comments up-to-date. **DO** use comments to tell what a section of code does. **DON'T** use comments for self-explanatory code.

Exercises

1. Write a program that writes I love C++ to the screen.

2. Write the smallest program that can be compiled, linked, and run.

3. BUG BUSTERS: Enter this program and compile it. Why does it fail? How can you fix it?

```
#include <iostream.h>
void main()
{
      cout << Is there a bug here?";
}
```

4. Fix the bug in Exercise 3 and recompile, link, and run it.

COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

INTRODUCTION TO COMPUTING

Unit 13: Variables and Constants

Introduction

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate that data. In this unit you will learn

- How to declare and define variables and constants.
- How to assign values to variables and manipulate those values.
- How to write the value of a variable to the screen.

What Is a Variable?

In C++ a variable is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

Your computer's memory can be viewed as a series of cubbyholes. Each cubbyhole is one of many, many such holes all lined up. Each cubbyhole--or memory location--is numbered sequentially. These numbers are known as memory addresses. A variable reserves one or more cubbyholes in which you may store a value.

Your variable's name (for example, `myVariable`) is a label on one of these cubbyholes, so that you can find it easily without knowing its actual memory address. Figure 3.1 is a schematic representation of this idea. As you can see from the figure, `myVariable` starts at memory address `103`. Depending on the size of `myVariable`, it can take up one or more memory addresses.

NOTE: RAM is random access memory. When you run your program, it is loaded into RAM from the disk file. All variables are also created in RAM. When programmers talk of memory, it is usually RAM to which they are referring.

Setting Aside Memory

When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable.

Each cubbyhole is one byte large. If the type of variable you create is two bytes in size, it needs two bytes of memory, or two cubbyholes. The type of the variable (for example, integer) tells the compiler how much memory (how many cubbyholes) to set aside for the variable.

Because computers use bits and bytes to represent values, and because memory is measured in bytes, it is important that you understand and are comfortable with these concepts. For a full review of this topic, please read Appendix B, "C++ Keywords."

Size of Integers

On any one computer, each variable type takes up a single, unchanging amount of room. That is, an integer might be two bytes on one machine, and four on another, but on either computer it is always the same, day in and day out.

A `char` variable (used to hold characters) is most often one byte long. A `short` integer is two bytes on most computers, a `long` integer is usually four bytes, and an integer (without the keyword `short` or `long`) can be two or four bytes. Listing 3.1 should help you determine the exact size of these types on your computer.

New Term: A *character* is a single letter, number, or symbol that takes up one byte of memory.

Listing 3.1. Determining the size of variable types on your computer.

```
1:   #include <iostream.h>
2:
3:   int main()
4:   {
5:     cout << "The size of an int is:\t\t"    << sizeof(int)    <<
" bytes.\n";
6:     cout << "The size of a short int is:\t" << sizeof(short)  <<
" bytes.\n";
7:     cout << "The size of a long int is:\t"  << sizeof(long)   <<
" bytes.\n";
8:     cout << "The size of a char is:\t\t"    << sizeof(char)   <<
" bytes.\n";
9:      cout << "The size of a float is:\t\t"   << sizeof(float)  << "
bytes.\n";
```

```
10:    cout << "The size of a double is:\t"    << sizeof(double) <<
" bytes.\n";
11:
12:        return 0;
13: }
```

Listing 3.1. (b) `Output:`

```
The size of an int is:         2 bytes.
The size of a short int is:      2 bytes.
The size of a long int is:       4 bytes.
The size of a char is:           1 bytes.
The size of a float is:          4 bytes.
The size of a double is:         8 bytes.
```

NOTE: On your computer, the number of bytes presented might be different.

Analysis: Most of Listing 3.1 should be pretty familiar. The one new feature is the use of the `sizeof()` function in lines 5 through 10. `sizeof()` is provided by your compiler, and it tells you the size of the object you pass in as a parameter. For example, on line 5 the keyword `int` is passed into `sizeof()`. Using `sizeof()`, I was able to determine that on my computer an `int` is equal to a `short int`, which is 2 bytes.

---

signed and unsigned

In addition, all integer types come in two varieties: `signed` and `unsigned`. The idea here is that sometimes you need negative numbers, and sometimes you don't. Integers (`short` and `long`) without the word "unsigned" are assumed to be `signed`. `Signed` integers are either negative or positive. `Unsigned` integers are always positive.

Because you have the same number of bytes for both `signed` and `unsigned` integers, the largest number you can store in an `unsigned` integer is twice as big as the largest positive number you can store in a `signed` integer. An `unsigned short` integer can handle numbers from 0 to 65,535. Half the numbers represented by a `signed short` are negative, thus a `signed short` can only represent numbers from -32,768 to 32,767. If this is confusing, be sure to read Appendix A, "Operator Precedence."

Fundamental Variable Types

Several other variable types are built into C++. They can be conveniently divided into integer variables (the type discussed so far), floating-point variables, and character variables.

Floating-point variables have values that can be expressed as fractions--that is, they are real numbers.

Character variables hold a single byte and are used for holding the 256 characters and symbols of the ASCII and extended ASCII character sets.

New Term: *The ASCII character set* is the set of characters standardized for use on computers. ASCII is an acronym for American Standard Code for Information Interchange. Nearly every computer operating system supports ASCII, though many support other international character sets as well.

The types of variables used in C++ programs are described in Table 3.1. This table shows the variable type, how much room this book assumes it takes in memory, and what kinds of values can be stored in these variables. The values that can be stored are determined by the size of the variable types, so check your output from Listing 3.1.

Table 3.1. Variable Types.

Type	*Size*	*Values*
`unsigned short int`	2 bytes	0 to 65,535
`short int`	2 bytes	-32,768 to 32,767
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long int`	4 bytes	-2,147,483,648 to 2,147,483,647
`int (16 bit)`	2 bytes	-32,768 to 32,767
`int (32 bit)`	4 bytes	-2,147,483,648 to 2,147,483,647
`unsigned int (16 bit)`	2 bytes	0 to 65,535
`unsigned int (32 bit)`	2 bytes	0 to 4,294,967,295
`char`	1 byte	256 character values
`float`	4 bytes	1.2e-38 to 3.4e38
`double`	8 bytes	2.2e-308 to 1.8e308

NOTE: The sizes of variables might be different from those shown in Table 3.1, depending on the compiler and the computer you are using. If your computer had the same output as was presented in Listing 3.1, Table 3.1 should apply to your compiler. If your output from Listing 3.1 was different, you should consult your compiler's manual for the values that your variable types can hold.

Defining a Variable

You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces. Legal variable names include `x`, `J23qrsnf`, and `myAge`. Good variable names tell you what the variables are for; using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called `myAge`: `int myAge;`

As a general programming practice, avoid such horrific names as `J23qrsnf`, and restrict singleletter variable names (such as `x` or `i`) to variables that are used only very briefly. Try to use expressive names such as `myAge` or `howMany`. Such names are easier to understand three weeks later when you are scratching your head trying to figure out what you meant when you wrote that line of code.

Try this experiment: Guess what these pieces of programs do, based on the first few lines of code:

Example 1

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

Example 2

```
main () {
     unsigned short Width;
     unsigned short Length;
     unsigned short Area;
     Area = Width * Length;
}
```

Clearly, the second program is easier to understand, and the inconvenience of having to type the longer variable names is more than made up for by how much easier it is to maintain the second program.

Case Sensitivity

C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named `age` is different from `Age`, which is different from `AGE`.

NOTE: Some compilers allow you to turn case sensitivity off. Don't be tempted to do this; your programs won't work with other compilers, and other C++ programmers will be very confused by your code.

There are various conventions for how to name variables, and although it doesn't much matter which method you adopt, it is important to be consistent throughout your program.

Many programmers prefer to use all lowercase letters for their variable names. If the name requires two words (for example, my car), there are two popular conventions: `my_car` or `myCar`. The latter form is called camel-notation, because the capitalization looks something like a camel's hump.

Some people find the underscore character (`my_car`) to be easier to read, while others prefer to avoid the underscore, because it is more difficult to type. This book uses camel-notation, in which the second and all subsequent words are capitalized: `myCar`, `theQuickBrownFox`, and so forth.

NOTE: Many advanced programmers employ a notation style that is often referred to as Hungarian notation. The idea behind Hungarian notation is to prefix every variable with a set of characters that describes its type. Integer variables might begin with a lowercase letter i, longs might begin with a lowercase l. Other notations indicate constants, globals, pointers, and so forth. Most of this is much more important in C programming, because C++ supports the creation of user-defined types (see Day 6, "Basic Classes") and because C++ is strongly typed.

Keywords

Some words are reserved by C++, and you may not use them as variable names. These are keywords used by the compiler to control your program. Keywords include `if`, `while`, `for`, and `main`. Your compiler manual should provide a complete list, but generally, any reasonable name for a variable is almost certainly not a keyword.

DO define a variable by writing the type, then the variable name. **DO** use meaningful variable names. DO remember that C++ is case sensitive. **DON'T** use C++ keywords as variable names. DO understand the number of bytes each variable type consumes in

memory, and what values can be stored in variables of that type. **DON'T** use `unsigned` variables for negative numbers.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example:

```
unsigned int myAge, myWeight;    // two unsigned int variables long
area, width, length;         // three longs
```

As you can see, `myAge` and `myWeight` are each declared as `unsigned` integer variables. The second line declares three individual `long` variables named `area`, `width`, and `length`. The type (`long`) is assigned to all the variables, so you cannot mix types in one definition statement.

Assigning Values to Your Variables

You assign a value to a variable by using the assignment operator (=). Thus, you would assign `5` to `Width` by writing

```
unsigned short Width;
Width = 5;
```

You can combine these steps and initialize `Width` when you define it by writing

```
unsigned short Width = 5;
```

Initialization looks very much like assignment, and with integer variables, the difference is minor. Later, when constants are covered, you will see that some values must be initialized because they cannot be assigned to. The essential difference is that initialization takes place at the moment you create the variable.

Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example:

```
// create two long variables and initialize them
long width = 5, length = 7;
```

This example initializes the `long` integer variable `width` to the value `5` and the `long` integer variable `length` to the value `7`. You can even mix definitions and initializations:

```
int myAge = 39, yourAge, hisAge = 40;
```

This example creates three type `int` variables, and it initializes the first and third.

Listing 3.2 shows a complete program, ready to compile, that computes the area of a rectangle and writes the answer to the screen.

Listing 3.2. A demonstration of the use of variables.

```
1:    // Demonstration of variables
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       unsigned short int Width = 5, Length;
7:       Length = 10;
8:
9:       // create  an unsigned short and initialize with result
10:         // of multiplying Width by Length
11:      unsigned short int Area  = Width * Length;
12:
13:      cout << "Width:" << Width << "\n";
14:      cout << "Length: "  << Length << endl;
15:      cout << "Area: " << Area << endl;
16:         return 0;
17: }

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

Analysis: Line 2 includes the required `include` statement for the `iostream`'s library so that `cout` will work. Line 4 begins the program.

On line 6, `Width` is defined as an `unsigned short` integer, and its value is initialized to `5`. Another `unsigned short` integer, `Length`, is also defined, but it is not initialized. On line 7, the value `10` is assigned to `Length`.

On line 11, an `unsigned short` integer, `Area`, is defined, and it is initialized with the value obtained by multiplying `Width` times `Length`. On lines 13-15, the values of the variables are printed to the screen. Note that the special word `endl` creates a new line.

When to Use short and When to Use long

One source of confusion for new C++ programmers is when to declare a variable to be type `long` and when to declare it to be type `short`. The rule, when understood, is fairly straightforward: If there is any chance that the value you'll want to put into your variable will be too big for its type, use a larger type.

As seen in Table 3.1, `unsigned short` integers, assuming that they are two bytes, can hold a value only up to 65,535. `Signed short` integers can hold only half that. Although `unsigned long` integers can hold an extremely large number (4,294,967,295) that is still quite finite. If you need a larger number, you'll have to go to `float` or `double`, and then you lose some precision. Floats and doubles can hold extremely large numbers, but only the first 7 or 19 digits are significant on most computers. That means that the number is rounded off after that many digits. **Wrapping Around an unsigned Integer**

The fact that `unsigned long` integers have a limit to the values they can hold is only rarely a problem, but what happens if you do run out of room?

When an `unsigned` integer reaches its maximum value, it wraps around and starts over, much as a car odometer might. Listing 3.4 shows what happens if you try to put too large a value into a `short` integer.

Listing 3.4.A demonstration of putting too large a value in an unsigned integer.

```
1: #include <iostream.h>
2:  int main()
3:  {
4:     unsigned short int smallNumber;
5:     smallNumber = 65535;
6:     cout << "small number:" << smallNumber << endl;
7:     smallNumber++;
8:     cout << "small number:" << smallNumber << endl;
9:     smallNumber++;
10:    cout << "small number:" << smallNumber << endl;
11:        return 0;
12: }
Output: small number:65535
small
number:0
small
number:1
```

Analysis: On line 4, `smallNumber` is declared to be an `unsigned short int`, which on my computer is a two-byte variable, able to hold a value between 0 and 65,535. On line 5, the maximum value is assigned to `smallNumber`, and it is printed on line 6.

On line 7, `smallNumber` is incremented; that is, 1 is added to it. The symbol for incrementing is `++` (as in the name C++--an incremental increase from C). Thus, the value in `smallNumber` would be `65,536`. However, `unsigned short` integers can't hold a number larger than 65,535, so the value is wrapped around to `0`, which is printed on line 8.

On line 9 `smallNumber` is incremented again, and then its new value, `1`, is printed.

Wrapping Around a signed Integer

A `signed` integer is different from an `unsigned` integer, in that half of the values you can represent are negative. Instead of picturing a traditional car odometer, you might picture one that rotates up for positive numbers and down for negative numbers. One mile from 0 is either 1 or -1. When you run out of positive numbers, you run right into the largest negative numbers and then count back down to 0. Listing 3.5 shows what happens when you add 1 to the maximum positive number in an `unsigned short` integer.

Listing 3.5. A demonstration of adding too large a number to a signed integer.

```
1:  #include <iostream.h>
2:  int main()
3:  {
4:     short int smallNumber;
5:     smallNumber = 32767;
6:     cout << "small number:" << smallNumber << endl;
7:     smallNumber++;
8:     cout << "small number:" << smallNumber << endl;
9:     smallNumber++;
10:    cout << "small number:" << smallNumber << endl;
11:        return 0;
12: }
```

```
Output:          small
number:32767     small
number:-32768    small
number:-32767
```

Analysis: On line 4, `smallNumber` is declared this time to be a `signed short` integer (if you don't explicitly say that it is `unsigned`, it is assumed to be `signed`). The program proceeds much as the preceding one, but the output is quite different. To fully understand this output, you must be comfortable with how `signed` numbers are represented as bits in a two-byte integer. For details, check Appendix C, "Binary and Hexadecimal."

The bottom line, however, is that just like an `unsigned` integer, the `signed` integer wraps around from its highest positive value to its highest negative value.

Characters

Character variables (`type char`) are typically 1 byte, enough to hold 256 values (see Appendix C). A `char` can be interpreted as a small number (0-255) or as a member of the ASCII set. ASCII stands for the American Standard Code for Information Interchange. The ASCII character set and its ISO (International Standards Organization) equivalent are a way to encode all the letters, numerals, and punctuation marks.

Computers do not know about letters, punctuation, or sentences. All they understand are numbers. In fact, all they really know about is whether or not a sufficient amount of electricity is at a particular junction of wires. If so, it is represented internally as a `1`; if not, it is represented as a `0`. By grouping ones and zeros, the computer is able to generate patterns that can be interpreted as numbers, and these in turn can be assigned to letters and punctuation.

In the ASCII code, the lowercase letter "a" is assigned the value `97`. All the lower- and uppercase letters, all the numerals, and all the punctuation marks are assigned values between 1 and 128. Another 128 marks and symbols are reserved for use by the computer maker, although the IBM extended character set has become something of a standard.

Characters and Numbers

When you put a character, for example, `` `a' ``, into a `char` variable, what is really there is just a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters (represented by a single quotation mark and then a letter, numeral, or punctuation mark, followed by a closing single quotation mark) and one of the ASCII values.

The value/letter relationship is arbitrary; there is no particular reason that the lowercase "a" is assigned the value `97`. As long as everyone (your keyboard, compiler, and screen) agrees, there is no problem. It is important to realize, however, that there is a big difference between the value `5` and the character `` `5' ``. The latter is actually valued at `53`, much as the letter `` `a' `` is valued at `97`.

Listing 3.6. Printing characters based on numbers

```
1:    #include <iostream.h>
2:    int main()
3:    {
4:    for (int i = 32; i<128; i++)
5:          cout << (char) i;
6:          return 0;
7: }
```

```
 Output: !"#$%G'()*+,./0123456789:;<>?@ABCDEFGHIJKLMNOP
_QRSTUVWXYZ[\]^'abcdefghijklmnopqrstuvwxyz<|>~s
```

This simple program prints the character values for the integers 32 through 127.

Special Printing Characters

The C++ compiler recognizes some special characters for formatting. Table 3.2 shows the most common ones. You put these into your code by typing the backslash (called the escape character), followed by the character. Thus, to put a tab character into your code, you would enter a single quotation mark, the slash, the letter t, and then a closing single quotation mark: `char tabCharacter = '\t';`

This example declares a `char` variable (`tabCharacter`) and initializes it with the character value `\t`, which is recognized as a tab. The special printing characters are used when printing either to the screen or to a file or other output device.

New Term: An *escape character* changes the meaning of the character that follows it. For example, normally the character `n` means the letter n, but when it is preceded by the escape character (`\`) it means new line.

Table 3.2. The Escape Characters.

Character	*What it means*
`\n`	new line
`\t`	tab
`\b`	backspace
`\"`	double quote
`\'`	single quote
`\?`	question mark
`\\`	backslash

Constants

Like variables, constants are data storage locations. Unlike variables, and as the name implies, constants don't change. You must initialize a constant when you create it, and you cannot assign a new value later.

Literal Constants

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 `int`

```
myAge = 39;
```

`myAge` is a variable of type `int`; `39` is a literal constant. You can't assign a value to `39`, and its value can't be changed.

Symbolic Constants

A symbolic constant is a constant that is represented by a name, just as a variable is represented.

Unlike a variable, however, after a constant is initialized, its value can't be changed.

If your program has one integer variable named `students` and another named `classes`, you could compute how many students you have, given a known number of classes, if you knew there were 15 students per class:

```
students = classes * 15;
```

NOTE: `*` indicates multiplication.

In this example, `15` is a literal constant. Your code would be easier to read, and easier to maintain, if you substituted a symbolic constant for this value:

```
students = classes * studentsPerClass
```

If you later decided to change the number of students in each class, you could do so where you define the constant `studentsPerClass` without having to make a change every place you used that value.

There are two ways to declare a symbolic constant in C++. The old, traditional, and now obsolete way is with a preprocessor directive, `#define`. Defining Constants with #define To define a constant the traditional way, you would enter this:

```
#define studentsPerClass 15
```

Note that `studentsPerClass` is of no particular type (`int`, `char`, and so on). `#define` does a simple text substitution. Every time the preprocessor sees the word `studentsPerClass`, it puts in the text `15`.

Because the preprocessor runs before the compiler, your compiler never sees your constant; it sees the number `15`. Defining Constants with const Although `#define` works, there is a new, much better way to define constants in C++:

```
const unsigned short int studentsPerClass = 15;
```

This example also declares a symbolic constant named `studentsPerClass`, but this time `studentsPerClass` is typed as an `unsigned short int`. This method has several advantages in making your code easier to maintain and in preventing bugs. The biggest difference is that this constant has a type, and the compiler can enforce that it is used according to its type.

NOTE: Constants cannot be changed while the program is running. If you need to change `studentsPerClass`, for example, you need to change the code and recompile.

DON'T use the term `int`. Use `short` and `long` to make it clear which size number you intended. **DO** watch for numbers overrunning the size of the integer and wrapping around incorrect values. **DO** give your variables meaningful names that reflect their use. **DON'T** use keywords as variable names.

Summary

This unit has discussed numeric and character variables and constants, which are used by C++ to store data during the execution of your program. Numeric variables are either integral (`char`, `short`, and `long int`) or they are floating point (`float` and `double`). Numeric variables can also be `signed` or `unsigned`. Although all the types can be of various sizes among different computers, the type specifies an exact size on any given computer.

You must declare a variable before it can be used, and then you must store the type of data that you've declared as correct for that variable. If you put too large a number into an integral variable, it wraps around and produces an incorrect result.

This unit also reviewed literal and symbolic constants, and showed two ways to declare a symbolic constant: using `#define` and using the keyword `const`.

Exercises

1. What would be the correct variable type in which to store the following information? **a.** Your age.

b. The area of your backyard.

c. The number of stars in the galaxy.

d. The average rainfall for the month of January.

2. Create good variable names for this information.

3. Declare a constant for pi as 3.14159.

4. Declare a `float` variable and initialize it using your pi constant.

COPPERBELT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

INTRODUCTION TO COMPUTING

Unit 14: Expressions and Statements

Introduction

At its heart, a program is a set of commands executed in sequence. The power in a program comes from its capability to execute one or another set of commands, based on whether a particular condition is true or false. In this unit, you will learn

- What statements are.
- What blocks are.
- What expressions are.
- How to branch your code based on conditions.
- What truth is, and how to act on it.

Statements

In C++ a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement). All C++ statements end with a semicolon, even the null statement, which is just the semicolon and nothing else. One of the most common statements is the following assignment statement:

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

Unlike in algebra, this statement does not mean that `x` equals `a+b`. This is read, "Assign the value of the sum of `a` and `b` to `x`," or "Assign to `x`, `a+b`." Even though this statement is doing two things, it is one statement and thus has one semicolon. The assignment operator assigns whatever is on the right side of the equal sign to whatever is on the left side.

New Term: A *null statement* is a statement that does nothing.

Whitespace

Whitespace (tabs, spaces, and newlines) is generally ignored in statements. The assignment statement previously discussed could be written as `x=a+b;`

or as

```
x                    =a

+           b          ;
```

Although this last variation is perfectly legal, it is also perfectly foolish. Whitespace can be used to make your programs more readable and easier to maintain, or it can be used to create horrific and indecipherable code. In this, as in all things, C++ provides the power; you supply the judgment.

New Term: *Whitespace characters* (spaces, tabs, and newlines) cannot be seen. If these characters are printed, you see only the white of the paper.

Blocks and Compound Statements

Any place you can put a single statement, you can put a compound statement, also called a block. A block begins with an opening brace ({) and ends with a closing brace (}). Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon. For example

```
{
      temp = a;
      a = b;
      b = temp;
}
```

This block of code acts as one statement and swaps the values in the variables `a` and `b`.

DO use a closing brace any time you have an opening brace. **DO** end your statements with a semicolon. **DO** use whitespace judiciously to make your code clearer.

Expressions

Anything that evaluates to a value is an expression in C++. An expression is said to return a value. Thus, `3+2;` returns the value `5` and so is an expression. All expressions are statements.

The myriad pieces of code that qualify as expressions might surprise you. Here are three examples:

```
3.2                        // returns the value 3.2
PI                         // float const that returns the value
3.14
SecondsPerMinute           // int const that returns 60
```

Assuming that PI is a constant equal to 3.14 and SecondsPerMinute is a constant equal to 60, all three of these statements are expressions.

The complicated expression

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

not only adds a and b and assigns the result to x, but returns the value of that assignment (the value of x) as well. Thus, this statement is also an expression. Because it is an expression, it can be on the right side of an assignment operator:

```
y = x = a + b;
```

This line is evaluated in the following order: Add a to b.

Assign the result of the expression a + b to x.

Assign the result of the assignment expression x = a + b to y.

If a, b, x, and y are all integers, and if a has the value 2 and b has the value 5, both x and y will be assigned the value 7.

Listing 4.1. Evaluating complex expressions.

```
1:    #include <iostream.h>
2:    int main()
3:    {
4:        int a=0, b=0, x=0, y=35;
5:        cout << "a: " << a << " b: " << b;
6:        cout << " x: " << x << " y: " << y << endl;
7:        a = 9;
8:        b = 7;
9:        y = x = a+b;
10:       cout << "a: " << a << " b: " << b;
11:       cout << " x: " << x << " y: " << y << endl;
12:          return 0;
13: }
Output: a: 0 b: 0 x: 0 y:
35 a: 9 b: 7 x: 16 y: 16
```

Analysis: On line 4, the four variables are declared and initialized. Their values are printed on lines 5 and 6. On line 7, a is assigned the value 9. One line 8, b is assigned the value 7. On line 9, the values of a and b are summed and the result is assigned to x. This expression (x = a+b) evaluates to a value (the sum of a + b), and that value is in turn assigned to y.

Operators

An operator is a symbol that causes the compiler to take an action. Operators act on operands, and in C++ all operands are expressions. In C++ there are several different categories of operators. Two of these categories are

- Assignment operators.
- Mathematical operators.

Assignment Operator

The assignment operator (=) causes the operand on the left side of the assignment operator to have its value changed to the value on the right side of the assignment operator. The expression `x = a + b;`

assigns the value that is the result of adding `a` and `b` to the operand `x`.

An operand that legally can be on the left side of an assignment operator is called an lvalue. That which can be on the right side is called (you guessed it) an rvalue.

Constants are r-values. They cannot be l-values. Thus, you can write

```
x = 35;           // ok
```

but you can't legally write

```
35 = x;           // error, not an lvalue!
```

New Term: An *lvalue* is an operand that can be on the left side of an expression. An rvalue is an operand that can be on the right side of an expression. Note that all l-values are r-values, but not all r-values are l-values. An example of an rvalue that is not an lvalue is a literal. Thus, you can write `x = 5;`, but you cannot write `5 = x;`.

Mathematical Operators

There are five mathematical operators: addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).

Addition and subtraction work as you would expect, although subtraction with `unsigned` integers can lead to surprising results, if the result is a negative number. You saw something much like this yesterday,

when variable overflow was described. Listing 4.2 shows what happens when you subtract a large `unsigned` number from a small `unsigned` number.

Listing 4.2. A demonstration of subtraction and integer overflow.

```
1: // Listing 4.2 - demonstrates subtraction and
2: // integer overflow
3: #include <iostream.h>
4:
5: int main()
6: {
7:    unsigned int difference;
8:    unsigned int bigNumber = 100;
9:    unsigned int smallNumber = 50;
10:   difference = bigNumber - smallNumber;
11:   cout << "Difference is: " << difference;
12:   difference = smallNumber - bigNumber;
13:   cout << "\nNow difference is: " << difference <<endl;
14:       return 0;
15: }
Output: Difference is: 50
Now difference is: 4294967246
```

Analysis: The subtraction operator is invoked on line 10, and the result is printed on line 11, much as we might expect. The subtraction operator is called again on line 12, but this time a large `unsigned` number is subtracted from a small `unsigned` number. The result would be negative, but because it is evaluated (and printed) as an `unsigned` number, the result is an overflow, as described yesterday. This topic is reviewed in detail in Appendix A, "Operator Precedence."

Integer Division and Modulus

Integer division is somewhat different from everyday division. When you divide 21 by 4, the result is a real number (a number with a fraction). Integers don't have fractions, and so the "remainder" is lopped off. The answer is therefore 5. To get the remainder, you take 21 modulus 4 (21 % 4) and the result is 1. The modulus operator tells you the remainder after an integer division.

Finding the modulus can be very useful. For example, you might want to print a statement on every 10th action. Any number whose value is 0 when you modulus 10 with that number is an exact multiple of 10. Thus 1 % 10 is 1, 2 % 10 is 2, and so forth, until 10 % 10, whose result is 0. 11 % 10 is back to 1, and this pattern continues until the next multiple of 10, which is 20. We'll use this technique when looping is discussed on Day 7, "More Program Flow."

WARNING: Many novice C++ programmers inadvertently put a semicolon after their `if` statements:

```
if(SomeValue    <    10);
SomeValue = 10;
```

What was intended here was to test whether `SomeValue` is less than 10, and if so, to set it to `10`, making `10` the minimum value for `SomeValue`. Running this code snippet will show that `SomeValue` is always set to `10`! Why? The `if` statement terminates with the semicolon (the do-nothing operator). Remember that indentation has no meaning to the compiler. This snippet could more accurately have been written as:

```
if (SomeValue < 10)   // test
;   // do nothing
SomeValue = 10;   // assign
```

Removing the semicolon will make the final line part of the `if` statement and will make this code do what was intended.

Combining the Assignment and Mathematical Operators

It is not uncommon to want to add a value to a variable, and then to assign the result back into the variable. If you have a variable `myAge` and you want to increase the value by two, you can write `int myAge = 5; int temp;`

```
temp = myAge + 2;   // add 5 + 2 and put it in temp myAge
= temp;              // put it back in myAge
```

This method, however, is terribly convoluted and wasteful. In C++, you can put the same variable on both sides of the assignment operator, and thus the preceding becomes `myAge = myAge + 2;`

which is much better. In algebra this expression would be meaningless, but in C++ it is read as "add two to the value in `myAge` and assign the result to `myAge`."

Even simpler to write, but perhaps a bit harder to read is

```
myAge += 2;
```

The self-assigned addition operator (+=) adds the rvalue to the lvalue and then reassigns the result into the lvalue. This operator is pronounced "plus-equals." The statement would be read "`myAge` plusequals two." If `myAge` had the value `4` to start, it would have `6` after this statement.

There are self-assigned subtraction (`-=`), division (`/=`), multiplication (`*=`), and modulus (`%=`) operators as well.

Increment and Decrement

The most common value to add (or subtract) and then reassign into a variable is `1`. In C++, increasing a value by 1 is called incrementing, and decreasing by 1 is called decrementing. There are special operators to perform these actions.

The increment operator (`++`) increases the value of the variable by 1, and the decrement operator (`--`) decreases it by 1. Thus, if you have a variable, `C`, and you want to increment it, you would use this statement:

```
C++;                // Start with C and increment it.
```

This statement is equivalent to the more verbose statement

```
C = C + 1;
```

which you learned is also equivalent to the moderately verbose statement

```
C += 1;
```

Prefix and Postfix

Both the increment operator (`++`) and the decrement operator(`--`) come in two varieties: prefix and postfix. The prefix variety is written before the variable name (`++myAge`); the postfix variety is written after (`myAge++`).

In a simple statement, it doesn't much matter which you use, but in a complex statement, when you are incrementing (or decrementing) a variable and then assigning the result to another variable, it matters very much. The prefix operator is evaluated before the assignment, the postfix is evaluated after.

The semantics of prefix is this: Increment the value and then fetch it. The semantics of postfix is different: Fetch the value and then increment the original.

This can be confusing at first, but if `x` is an integer whose value is `5` and you write `int`

```
a = ++x;
```

you have told the compiler to increment `x` (making it `6`) and then fetch that value and assign it to `a`. Thus, `a` is now `6` and `x` is now `6`.

If, after doing this, you write

```
int b = x++;
```

you have now told the compiler to fetch the value in x (6) and assign it to b, and then go back and increment x. Thus, b is now 6, but x is now 7. Listing 4.3 shows the use and implications of both types.

Listing 4.3. A demonstration of prefix and postfix operators.

```
1:  // Listing 4.3 - demonstrates use of
2:  // prefix and postfix increment and
3:  // decrement operators
4:  #include <iostream.h>
5:  int main()
6:  {
7:       int myAge = 39;      // initialize two integers
8:       int yourAge = 39;
9:       cout << "I am: " << myAge << " years old.\n";
10:      cout << "You are: " << yourAge << " years old\n";
11:      myAge++;         // postfix increment
12:      ++yourAge;       // prefix increment
13:      cout << "One year passes...\n";
14:      cout << "I am: " << myAge << " years old.\n";
15:      cout << "You are: " << yourAge << " years old\n";
16:      cout << "Another year passes\n";
17:      cout << "I am: " << myAge++ << " years old.\n";
18:      cout << "You are: " << ++yourAge << " years old\n";
19:      cout << "Let's print it again.\n";
20:      cout << "I am: " << myAge << " years old.\n";
21:      cout << "You are: " << yourAge << " years old\n";
22:        return 0;
23: }
Output: I am      39 years old
You are   39 years old
One year passes
I am      40 years old
You are   40 years old
Another year passes
I am      40 years old
You are   41 years old
Let's print it again
I am      41 years old
You are   41 years old
```

Analysis: On lines 7 and 8, two integer variables are declared, and each is initialized with the value 39. Their values are printed on lines 9 and 10.

On line 11, myAge is incremented using the postfix increment operator, and on line 12, yourAge is incremented using the prefix increment operator. The results are printed on lines 14 and 15, and they are identical (both 40).

On line 17, `myAge` is incremented as part of the printing statement, using the postfix increment operator. Because it is postfix, the increment happens after the print, and so the value `40` is printed again. In contrast, on line 18, `yourAge` is incremented using the prefix increment operator. Thus, it is incremented before being printed, and the value displays as `41`.

Finally, on lines 20 and 21, the values are printed again. Because the increment statement has completed, the value in `myAge` is now `41`, as is the value in `yourAge`. **Precedence**

In the complex statement

```
x = 5 + 3 * 8;
```

which is performed first, the addition or the multiplication? If the addition is performed first, the answer is 8 * 8, or 64. If the multiplication is performed first, the answer is 5 + 24, or 29.

Every operator has a precedence value, and the complete list is shown in Appendix A, "Operator Precedence." Multiplication has higher precedence than addition, and thus the value of the expression is 29.

When two mathematical operators have the same precedence, they are performed in left-to-right order. Thus

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

is evaluated multiplication first, left to right. Thus, 8*9 = 72, and 6*4 = 24. Now the expression is essentially

```
x = 5 + 3 + 72 + 24;
```

Now the addition, left to right, is 5 + 3 = 8; 8 + 72 = 80; 80 + 24 = 104.

Be careful with this. Some operators, such as assignment, are evaluated in right-to-left order! In any case, what if the precedence order doesn't meet your needs? Consider the expression

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

In this expression, you do not want to multiply the `NumMinutesToType` variable by 60 and then add it to `NumMinutesToThink`. You want to add the two variables to get the total number of minutes, and then you want to multiply that number by 60 to get the total seconds.

In this case, you use parentheses to change the precedence order. Items in parentheses are evaluated at a higher precedence than any of the mathematical operators. Thus

```
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60
```

will accomplish what you want.

Nesting Parentheses

For complex expressions, you might need to nest parentheses one within another. For example, you might need to compute the total seconds and then compute the total number of people who are involved before multiplying seconds times people:

```
TotalPersonSeconds = (( (NumMinutesToThink + NumMinutesToType)*
60) *(PeopleInTheOffice + PeopleOnVacation))
```

This complicated expression is read from the inside out. First, `NumMinutesToThink` is added to `NumMinutesToType`, because these are in the innermost parentheses. Then this sum is multiplied by 60. Next, `PeopleInTheOffice` is added to `PeopleOnVacation`. Finally, the total number of people found is multiplied by the total number of seconds.

This example raises an important related issue. This expression is easy for a computer to understand, but very difficult for a human to read, understand, or modify. Here is the same expression rewritten, using some temporary integer variables:

```
TotalMinutes = NumMinutesToThink + NumMinutesToType;
TotalSeconds = TotalMinutes * 60;
TotalPeople = PeopleInTheOffice + PeopleOnVacation;
TotalPersonSeconds = TotalPeople * TotalSeconds;
```

This example takes longer to write and uses more temporary variables than the preceding example, but it is far easier to understand. Add a comment at the top to explain what this code does, and change the `60` to a symbolic constant. You then will have code that is easy to understand and maintain.

DO remember that expressions have a value. **DO** use the prefix operator (++variable) to increment or decrement the variable before it is used in the expression. **DO** use the postfix operator (variable++) to increment or decrement the variable after it is used. **DO** use parentheses to change the order of precedence. **DON'T** nest too deeply, because the expression becomes hard to understand and maintain.

The Nature of Truth

In C++, zero is considered false, and all other values are considered true, although true is usually represented by 1. Thus, if an expression is false, it is equal to zero, and if an expression is equal to zero, it is false. If a statement is true, all you know is that it is nonzero, and any nonzero statement is true.

Relational Operators

The relational operators are used to determine whether two numbers are equal, or if one is greater or less than the other. Every relational statement evaluates to either `1` (`TRUE`) or `0` (`FALSE`). The relational operators are presented later, in Table 4.1.

If the integer variable `myAge` has the value `39`, and the integer variable `yourAge` has the value `40`, you can determine whether they are equal by using the relational "equals" operator: `myAge == yourAge; // is the value in myAge the same as in yourAge?`

This expression evaluates to `0`, or `false`, because the variables are not equal. The expression `myAge > yourAge;  // is myAge greater than yourAge?`

evaluates to `0` or `false`.

WARNING: Many novice C++ programmers confuse the assignment operator (=) with the equals operator (==). This can create a nasty bug in your program.

There are six relational operators: equals (==), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), and not equals (!=). Table 4.1 shows each relational operator, its use, and a sample code use.

Table 4.1. The Relational Operators.

Name	*Operator*	*Sample*	*Evaluates*
Equals	==	`100 == 50;`	`false`
		`50 == 50;`	`true`
Not Equals	!=	`100 != 50;`	`true`
		`50 != 50;`	`false`
Greater Than	>	`100 > 50;`	`true`
		`50 > 50;`	`false`
Greater Than or Equals	>=	`100 >= 50;`	`true`
		`50 >= 50;`	`true`
Less Than	<	`100 < 50;`	`false`
		`50 < 50;`	`false`
Less Than or Equals	<=	`100 <= 50;`	`false`
		`50 <= 50;`	`true`

DO remember that relational operators return the value `1` (`true`) or `0` (`false`).

DON'T confuse the assignment operator (=) with the equals relational operator (==). This is one of the most common C++ programming mistakes--be on guard for it.

The if Statement

Normally, your program flows along line by line in the order in which it appears in your source code. The `if` statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.

The simplest form of an `if` statement

is this:

```
if (expression)
     statement;
```

The expression in the parentheses can be any expression at all, but it usually contains one of the relational expressions. If the expression has the value `0`, it is considered false, and the statement is skipped. If it has any nonzero value, it is considered true, and the statement is executed. Consider the following example:

```
if (bigNumber > smallNumber)
     bigNumber = smallNumber;
```

This code compares `bigNumber` and `smallNumber`. If `bigNumber` is larger, the second line sets its value to the value of `smallNumber`.

Because a block of statements surrounded by braces is exactly equivalent to a single statement, the following type of branch can be quite large and powerful:

```
if (expression)
{
     statement1;
     statement2;
     statement3;
}
```

Here's a simple example of this usage:

```
if (bigNumber > smallNumber)
{
     bigNumber = smallNumber;
     cout << "bigNumber: " << bigNumber << "\n";
     cout << "smallNumber: " << smallNumber << "\n";
}
```

This time, if `bigNumber` is larger than `smallNumber`, not only is it set to the value of `smallNumber`, but an informational message is printed. Listing 4.4 shows a more detailed example of branching based on relational operators.

Listing 4.4. A demonstration of branching based on relational operators.

```
1:  // Listing 4.4 - demonstrates if statement
2:  // used with relational operators
3:  #include <iostream.h>
4:  int main()
5:  {
6:        int RedSoxScore, YankeesScore;
7:        cout << "Enter the score for the Red Sox: ";
8:        cin >> RedSoxScore;
9:
10:       cout << "\nEnter the score for the Yankees: ";
11:       cin >> YankeesScore;
12:
13:       cout << "\n";
14:
15:       if (RedSoxScore > YankeesScore)
16:            cout << "Go Sox!\n";
17:
18:       if (RedSoxScore < YankeesScore)
19:       {
20:            cout << "Go Yankees!\n";
21:           cout << "Happy days in New York!\n"; 22:
          }
23:
24:       if (RedSoxScore == YankeesScore)
25:       {
26:            cout << "A tie? Naah, can't be.\n";
27:            cout << "Give me the real score for the Yanks: ";
28:            cin >> YankeesScore;
29:
30:            if (RedSoxScore > YankeesScore)
31:                 cout << "Knew it! Go Sox!";
32:
33:            if (YankeesScore > RedSoxScore)
34:                 cout << "Knew it! Go Yanks!";
35:
36:            if (YankeesScore == RedSoxScore)
37:                 cout << "Wow, it really was a tie!"; 38:
            }
39:
40:         cout << "\nThanks for telling me.\n";
41:         return 0;
42:     }
```

```
Output: Enter the score for the Red Sox: 10 Enter

the score for the Yankees: 10

A tie? Naah, can't be
Give me the real score for the Yanks: 8
Knew it! Go Sox!
Thanks        for
telling me.
```

Analysis: This program asks for user input of scores for two baseball teams, which are stored in integer variables. The variables are compared in the `if` statement on lines 15, 18, and 24.

If one score is higher than the other, an informational message is printed. If the scores are equal, the block of code that begins on line 24 and ends on line 38 is entered. The second score is requested again, and then the scores are compared again.

Note that if the initial Yankees score was higher than the Red Sox score, the `if` statement on line 15 would evaluate as `FALSE`, and line 16 would not be invoked. The test on line 18 would evaluate as `true`, and the statements on lines 20 and 21 would be invoked. Then the `if` statement on line 24 would be tested, and this would be false (if line 18 was true). Thus, the program would skip the entire block, falling through to line 39.

In this example, getting a true result in one `if` statement does not stop other `if` statements from being tested.

Indentation Styles

Listing 4.3 shows one style of indenting `if` statements. Although there are dozens of variations, these appear to be the favorite three:

- Putting the initial brace after the condition and aligning the closing brace under the `if` to close the statement block.

```
if(expression){
     statements
}
```

- Aligning the braces under the if and indenting the statements.

```
if (expression)
{
     statements
}
```

- Indenting the braces and statements.

```
if (expression)
   {
   statements
   }
```

In the lectures, we use the middle alternative, because it easier to understand where blocks of statements begin and end if the braces line up with each other and with the condition being tested. Again, it doesn't matter much which style you choose, as long as you are consistent with it.

else

Often your program will want to take one branch if your condition is true, another if it is false. In Listing 4.3, you wanted to print one message (`Go Sox!`) if the first test (`RedSoxScore > Yankees`) evaluated `TRUE`, and another message (`Go Yanks!`) if it evaluated `FALSE`.

The method shown so far, testing first one condition and then the other, works fine but is a bit cumbersome. The keyword `else` can make for far more readable code:

```
if (expression)
   statement;
else
   statement;
```

Listing 4.5 demonstrates the use of the keyword `else`.

Listing 4.5. Demonstrating the else keyword.

```
1:   // Listing 4.5 - demonstrates if statement
2:   // with else clause
3:   #include <iostream.h>
4:   int main()
5:   {
6:      int firstNumber, secondNumber;
7:      cout << "Please enter a big number: ";
8:      cin >> firstNumber;
9:      cout << "\nPlease enter a smaller number: ";
10:     cin >> secondNumber;
11:     if (firstNumber > secondNumber)
12:          cout << "\nThanks!\n";
13:     else
14:          cout << "\nOops. The second is bigger!";
15:
16:        return 0;
17: }
```

```
Output: Please enter a big number: 10

Please enter a smaller number: 12

Oops. The second is bigger!
```

Analysis: The `if` statement on line 11 is evaluated. If the condition is true, the statement on line 12 is run; if it is false, the statement on line 14 is run. If the `else` clause on line 13 were removed, the statement on line 14 would run whether or not the `if` statement was true. Remember, the `if` statement ends after line 12. If the `else` was not there, line 14 would just be the next line in the program.

Remember that either or both of these statements could be replaced with a block of code in braces.

The if Statement

The syntax for the `if` statement is as follows: Form 1

```
if (expression)
      statement;
next statement;
```

If the expression is evaluated as `TRUE`, the statement is executed and the program continues with the next statement. If the expression is not true, the statement is ignored and the program jumps to the next statement. Remember that the statement can be a single statement ending with a semicolon or a block enclosed in braces. Form 2

```
if (expression)
      statement1;
else
     statement2;
next statement;
```

If the expression evaluates `TRUE`, `statement1` is executed; otherwise, `statement2` is executed. Afterwards, the program continues with the next statement. Example 1

```
Example
if (SomeValue < 10)
     cout << "SomeValue is less than 10");
else
      cout << "SomeValue is not less than 10!");
cout << "Done." << endl;
```

Advanced if Statements

It is worth noting that any statement can be used in an `if` or `else` clause, even another `if` or `else` statement. Thus, you might see complex `if` statements in the following form:

```
if (expression1)
{
    if (expression2)
          statement1;
    else{
         if (expression3)
               statement2;
         else
               statement3;
    }
 }
else
     statement4;
```

This cumbersome `if` statement says, "If expression1 is true and expression2 is true, execute statement1. If expression1 is true but expression2 is not true, then if expression3 is true execute statement2. If expression1 is true but expression2 and expression3 are false, execute statement3. Finally, if expression1 is not true, execute statement4." As you can see, complex `if` statements can be confusing!

Listing 4.6 gives an example of such a complex `if` statement.

Listing 4.6. A complex, nested if statement.

```
1:  // Listing 4.5 - a complex nested
2:  // if statement
3:  #include <iostream.h>
4:  int main()
5:  {
6:      // Ask for two numbers
7:      // Assign the numbers to bigNumber and littleNumber
8:      // If bigNumber is bigger than littleNumber,
9:      // see if they are evenly divisible
10:     // If they are, see if they are the same number
11:
12:     int firstNumber, secondNumber;
13:     cout << "Enter two numbers.\nFirst: ";
14:     cin >> firstNumber;
15:     cout << "\nSecond: ";
16:     cin >> secondNumber;
17:     cout << "\n\n";
18:
19:     if (firstNumber >= secondNumber)
20:     {
21:      if ((firstNumber % secondNumber)== 0) //evenly divisible?
22:       {
23:            if (firstNumber == secondNumber)
24:                 cout << "They are the same!\n";
25:            else
```

```
26:                 cout << "They are evenly divisible!\n";
27:         }
28:         else
29:              cout << "They are not evenly divisible!\n";
30:      }
31:      else
32:        cout << "Hey! The second one is larger!\n";
33:     return 0;
34: }
```

```
Output: Enter two numbers.
First: 10
Second: 2

They are evenly divisible!
```

Analysis: Two numbers are prompted for one at a time, and then compared. The first `if` statement, on line 19, checks to ensure that the first number is greater than or equal to the second. If not, the `else` clause on line 31 is executed.

If the first `if` is true, the block of code beginning on line 20 is executed, and the second `if` statement is tested, on line 21. This checks to see whether the first number modulo the second number yields no remainder. If so, the numbers are either evenly divisible or equal. The `if` statement on line 23 checks for equality and displays the appropriate message either way.

If the `if` statement on line 21 fails, the `else` statement on line 28 is executed.

Using Braces in Nested if Statements

Although it is legal to leave out the braces on `if` statements that are only a single statement, and it is legal to nest `if` statements, such as

```
if (x > y)                // if x is bigger than y
     if (x < z)           // and if x is smaller than z
          x = y;          // then set x to the value in z
```

when writing large nested statements, this can cause enormous confusion. Remember, whitespace and indentation are a convenience for the programmer; they make no difference to the compiler. It is easy to confuse the logic and inadvertently assign an `else` statement to the wrong `if` statement. Listing 4.7 illustrates this problem.

Listing 4.7. A demonstration of why braces help clarify which else statement goes with which if statement.

```
1:   // Listing 4.7 - demonstrates why braces
2:   // are important in nested if statements
```

```
3:    #include <iostream.h>
4:    int main()
5:    {
6:      int x;
7:      cout << "Enter a number less than 10 or greater than 100: ";
8:      cin >> x;
9:      cout << "\n";
10:
11:      if (x > 10)
12:         if (x > 100)
13:              cout << "More than 100, Thanks!\n";
14:       else                              // not the else intended!
15:         cout << "Less than 10, Thanks!\n";
16:
17:          return 0;
18: }
```

```
Output: Enter a number less than 10 or greater than 100: 20 Less

than 10, Thanks!
```

Analysis: The programmer intended to ask for a number between 10 and 100, check for the correct value, and then print a thank-you note.

If the `if` statement on line 11 evaluates `TRUE`, the following statement (line 12) is executed. In this case, line 12 executes when the number entered is greater than 10. Line 12 contains an `if` statement also. This `if` statement evaluates `TRUE` if the number entered is greater than 100. If the number is not greater than 100, the statement on line 13 is executed.

If the number entered is less than or equal to 10, the `if` statement on line 10 evaluates to `FALSE`. Program control goes to the next line following the `if` statement, in this case line 16. If you enter a number less than 10, the output is as follows:

```
Enter a number less than 10 or greater than 100: 9
```

The `else` clause on line 14 was clearly intended to be attached to the `if` statement on line 11, and thus is indented accordingly. Unfortunately, the `else` statement is really attached to the `if` statement on line 12, and thus this program has a subtle bug.

It is a subtle bug because the compiler will not complain. This is a legal C++ program, but it just doesn't do what was intended. Further, most of the times the programmer tests this program, it will appear to work. As long as a number that is greater than 100 is entered, the program will seem to work just fine.

Listing 4.8 fixes the problem by putting in the necessary braces.

Listing 4.8. A demonstration of the proper use of braces with an if statement

```
1:    // Listing 4.8 - demonstrates proper use of braces
2:    // in nested if statements
3:    #include <iostream.h>
4:    int main()
5:    {
6:      int x;
7:      cout << "Enter a number less than 10 or greater than 100:
";
8:      cin >> x;
9:      cout << "\n";
10:
11:     if (x > 10)
12:     {
13:        if (x > 100)
14:             cout << "More than 100, Thanks!\n";
15:     }
16:     else                              // not the else intended!
17:        cout << "Less than 10, Thanks!\n";
18:         return 0;
19: }
Output: Enter a number less than 10 or greater than 100: 20
```

Analysis: The braces on lines 12 and 15 make everything between them into one statement, and now the `else` on line 16 applies to the `if` on line 11 as intended.

The user typed `20`, so the `if` statement on line 11 is true; however, the `if` statement on line 13 is false, so nothing is printed. It would be better if the programmer put another `else` clause after line 14 so that errors would be caught and a message printed.

NOTE: The programs shown in these notes are written to demonstrate the particular issues being discussed. They are kept intentionally simple; there is no attempt to "bulletproof" the code to protect against user error. In professional-quality code, every possible user error is anticipated and handled gracefully.

Logical Operators

Often you want to ask more than one relational question at a time. "Is it true that x is greater than y, and also true that y is greater than z?" A program might need to determine that both of these conditions are true, or that some other condition is true, in order to take an action.

Imagine a sophisticated alarm system that has this logic: "If the door alarm sounds AND it is after six p.m. AND it is NOT a holiday, OR if it is a weekend, then call the police." C++'s three logical operators are used to make this kind of evaluation. These operators are listed in Table 4.2.

Table 4.2. The Logical Operators.

Operator	*Symbol*	*Example*
AND	&&	expression1 && expression2
OR	||	expression1 || expression2
NOT	!	!expression

Logical AND

A logical AND statement evaluates two expressions, and if both expressions are true, the logical AND statement is true as well. If it is true that you are hungry, AND it is true that you have money, THEN it is true that you can buy lunch. Thus,

```
if ((x == 5) && (y == 5))
```

would evaluate TRUE if both x and y are equal to 5, and it would evaluate FALSE if either one is not equal to 5. Note that both sides must be true for the entire expression to be true.

Note that the logical AND is two && symbols. A single & symbol is a different operator, discussed on Day 21, "What's Next."

Logical OR

A logical OR statement evaluates two expressions. If either one is true, the expression is true. If you have money OR you have a credit card, you can pay the bill. You don't need both money and a credit card; you need only one, although having both would be fine as well. Thus, `if ( (x == 5) || (y == 5))` evaluates TRUE if either x or y is equal to 5, or if both are.

Note that the logical OR is two || symbols. A single | symbol is a different operator, discussed on Day 21.

Logical NOT

A logical NOT statement evaluates true if the expression being tested is false. Again, if the expression being tested is false, the value of the test is TRUE! Thus `if (!(x == 5))` is true only if x is not equal to 5. This is exactly the same as writing `if (x != 5)`

Relational Precedence

Relational operators and logical operators, being C++ expressions, each return a value: 1 (TRUE) or 0 (FALSE). Like all expressions, they have a precedence order (see Appendix A) that determines which relations are evaluated first. This fact is important when determining the value of the statement

```
if ( x > 5 &&  y > 5  || z > 5)
```

It might be that the programmer wanted this expression to evaluate TRUE if both x and y are greater than 5 or if z is greater than 5. On the other hand, the programmer might have wanted this expression to evaluate TRUE only if x is greater than 5 and if it is also true that either y is greater than 5 or z is greater than 5.

If x is 3, and y and z are both 10, the first interpretation will be true (z is greater than 5, so ignore x and y), but the second will be false (it isn't true that both x and y are greater than 5 nor is it true that z is greater than 5).

Although precedence will determine which relation is evaluated first, parentheses can both change the order and make the statement clearer:

```
if (  (x > 5)  && (y > 5 ||  z > 5) )
```

Using the values from earlier, this statement is false. Because it is not true that x is greater than 5, the left side of the AND statement fails, and thus the entire statement is false. Remember that an AND statement requires that both sides be true--something isn't both "good tasting" AND "good for you" if it isn't good tasting.

NOTE: It is often a good idea to use extra parentheses to clarify what you want to group. Remember, the goal is to write programs that work and that are easy to read and understand.

More About Truth and Falsehood

In C++, zero is false, and any other value is true. Because an expression always has a value, many C++ programmers take advantage of this feature in their if statements. A statement such as

```
if (x)           // if x is true (nonzero)
x = 0;
```

can be read as "If x has a nonzero value, set it to 0." This is a bit of a cheat; it would be clearer if written

```
if (x != 0)      // if x is nonzero
x = 0;
```

Both statements are legal, but the latter is clearer. It is good programming practice to reserve the former method for true tests of logic, rather than for testing for nonzero values.

These two statements also are equivalent:

```
if (!x)          // if x is false (zero)
if (x == 0)      // if x is zero
```

The second statement, however, is somewhat easier to understand and is more explicit.

DO put parentheses around your logical tests to make them clearer and to make the precedence explicit. **DO** use braces in nested `if` statements to make the `else` statements clearer and to avoid bugs. **DON'T** use `if(x)` as a synonym for `if(x != 0)`; the latter is clearer. **DON'T** use `if(!x)` as a synonym for `if(x == 0)`; the latter is clearer.

NOTE: It is common to define your own enumerated Boolean (logical) type with `enum Bool {FALSE, TRUE};`. This serves to set `FALSE` to `0` and `TRUE` to `1`.

Exercises

1. Write a single if statement that examines two integer variables and changes the larger to the smaller, using only one else clause.

2. Examine the following program. Imagine entering three numbers, and write what output you expect.

```
1: #include <iostream.h>
2: int main()
3: {
4:        int a, b, c;
5:        cout << "Please enter three numbers\n";
6:        cout << "a: ";
7:        cin >> a;
8:        cout << "\nb: ";
9:        cin >> b;
10:       cout << "\nc: ";
11:       cin >> c;
12:
13:       if (c = (a-b))
14:       {
               cout << "a: ";
15:            cout << a;
16:            cout << "minus b: ";
17:            cout << b;
18:            cout << "equals c: ";
19:            cout << c << endl;
          }
20:       else
21:            cout << "a-b does not equal c: " << endl;
```

```
22:       return 0;
23:  }
```

3. Enter the program from Exercise 2; compile, link, and run it. Enter the numbers 20, 10, and 50. Did you get the output you expected? Why not?

4. Examine this program and anticipate the output:

```
1: #include <iostream.h>
2: int main()
3: {
4: int a = 1, b = 1, c;
5: if (c = (a-b))
6: cout << "The value of c is: " << c;
7: return 0;
8: }
```

5. Enter, compile, link, and run the program from Exercise 4. What was the output? Why?