Making Everything Easier!™

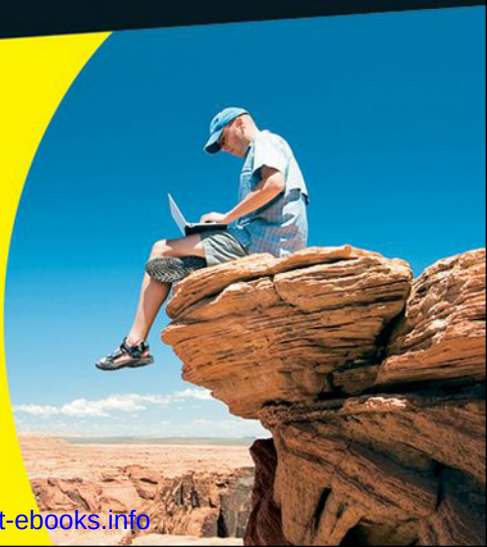
Beginning Programming with C++

FOR DUMMIES®

Learn to:

- Think like a programmer and understand how C++ works
- Create programs and get bugs out of your code
- Master basic development concepts and techniques in C++

Find source code from the book and the Code::Blocks C++ compiler on the companion CD-ROM



Stephen R. Davis
Author of C++ For Dummies

Beginning Programming with C++ FOR DUMMIES®

by Stephen R. Davis

Wiley Publishing, Inc.

Beginning Programming with C++ For Dummies®
Published by
Wiley Publishing, Inc.
111 River Street
Hoboken, NJ 07030-5774

www.wiley.com

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

Library of Congress Control Number: 2010930969

ISBN: 978-0-470-61797-7

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents at a Glance

Table of Contents

Introduction

Welcome to *Beginning Programming with C++ For Dummies.* This book is intended for the reader who wants to learn to program.

Somehow over the years, programming has become associated with mathematics and logic calculus and other complicated things. I never quite understood that. Programming is a skill like writing advertising or drawing or photography. It does require the ability to think a problem through, but I've known some really good programmers who had zero math skills. Some people are naturally good at it and pick it up quickly, others not so good and not so quick. Nevertheless, anyone with enough patience and "stick-to-itiveness" can learn to program a computer. Even me.

About Beginning Programming with C++ For Dummies

Learning to program necessarily means learning a programming language. This book is based upon the C++ programming language. A Windows version of the suggested compiler is included on the CD-ROM accompanying this book. Macintosh and Linux versions are available for download at `www.codeblocks.org`. (Don't worry: I include step-by-step instructions for how to install the package and build your first program in the book.)

The goal of this book is to teach you the basics of programming in C++, not to inundate you with every detail of the C++ programming language. At the end of this book, you will be able to write a reasonably sophisticated program in C++. You will also be in a position to quickly grasp a number of other similar languages, such as Java and C#.NET.

In this book, you will discover what a program is, how it works, plus how to do the following:

- Install the CodeBlocks C++ compiler and use it to build a program
- Create and evaluate expressions
- Direct the flow of control through your program

- Create data structures that better model the real world
- Define and use C++ pointers
- Manipulate character strings to generate the output the way you want to see it
- Write to and read from files

Foolish Assumptions

I try to make very few assumptions in this book about the reader, but I do assume the following:

- **You have a computer.** Most readers will have computers that run Windows; however, the programs in this book run equally well on Windows, Macintosh, Linux, and Unix. In fact, since C++ is a standardized language, these programs should run on any computer that has a C++ compiler.
- **You know the basics of how to use your computer.** For example, I assume that you know how to run a program, copy a file, create a folder, and so on.
- **You know how to navigate through menus.** I include lots of instructions like "Click on File and then Open." If you can follow that instruction, then you're good to go.
- **You are new to programming.** I don't assume that you know anything about programming. Heck, I don't even assume that you know what programming is.

Conventions Used in This Book

To help you navigate this book as efficiently as possible, I use a few conventions:

- C++ terms are in monofont typeface, `like this`.
- New terms are emphasized with *italics* (and defined).
- Numbered steps that you need to follow and characters you need to type are set in **bold**.

What You Don't Have to Read

I encourage you to read one part of the book; then put the book away and play for a while before moving to the next part. The book is organized so that by the end of each part, you have mastered enough new material to go out and write programs.

I'd like to add the following advice:

- If you already know what programming is but nothing about C++, you can skip Chapter 1.
- I recommend that you use the CodeBlocks compiler that comes with the book, even if you want to use a different C++ compiler after you finish the book. However, if you insist and don't want to use CodeBlocks, you can skip Chapter 2.
- Skim through Chapter 3 if you've already done a little computer programming.
- Start concentrating at Chapter 4, even if you have experience with other languages such as BASIC.
- You can stop reading after Chapter 20 if you're starting to feel saturated. Chapter 21 opens up the new topic of object-oriented programming — you don't want to take that on until you feel really comfortable with what you've learned so far.
- You can skip any of the TechnicalStuff icons.

How This Book Is Organized

Beginning Programming with C++ For Dummies is split into seven parts. You don't have to read it sequentially, and you don't even have to read all the sections in any particular chapter. You can use the Table of Contents and the Index to find the information you need and quickly get your answer. In this section, I briefly describe what you'll find in each part.

Part 1: Let's Get Started

This part describes what programs are and how they work. Using a fictitious tire-changing computer, I take you through several algorithms for removing a tire from a car to give you a feel for how programs work. You'll also get CodeBlocks up and running on your computer before leaving this part.

Part II: Writing a Program: Decisions, Decisions

This part introduces you to the basics of programming with C++. You will find out how to declare integer variables and how to write simple expressions. You'll even discover how to make decisions within a program, but you won't be much of an expert by the time you finish this part.

Part III: Becoming a Functional Programmer

Here you learn how to direct the flow of control within your programs. You'll find out how to loop, how to break your code into modules (and why), and how to build these separate modules back into a single program. At the end of this part, you'll be able to write real programs that actually solve problems.

Part IV: Data Structures

This part expands your knowledge of data types. Earlier sections of the book are limited to integers; in this part, you work with characters, decimals, and arrays; and you even get to define your own types. Finally, this is the part where you master the most dreaded topic, the C++ pointer.

Part V: Object-Oriented Programming

This is where you expand your knowledge into object-oriented techniques, the stuff that differentiates C++ from its predecessors, most notably C. (Don't worry if you don't know what object-oriented programming is — you aren't supposed to yet.) You'll want to be comfortable with the material in Parts I through IV before jumping into this part, but you'll be a much stronger programmer by the time you finish it.

Part VI: Advanced Strokes

This is a collection of topics that are important but that didn't fit in the earlier parts. For example, here's where I discuss how to create, read to, and write from files.

Part VII: The Part of Tens

This part includes lists of what to do (and what not to do) when programming to avoid creating bugs needlessly. In addition, this part includes some advice about what topics to study next, should you decide to expand your knowledge of C++.

The CD-ROM Appendix

This part describes what's on the enclosed CD-ROM and how to install it.

Icons Used in This Book

What's a *Dummies* book without icons pointing you in the direction of really great information that's sure to help you along your way? In this section, I briefly describe each icon I use in this book.

The Tip icon points out helpful information that is likely to make your job easier.

This icon marks a generally interesting and useful fact — something that you might want to remember for later use. I also use this icon to remind you of some fact that you may have skipped over in an earlier chapter.

The Warning icon highlights lurking danger. With this icon, I'm telling you to pay attention and proceed with caution.

When you see this icon, you know that there's techie stuff nearby. If you're not feeling very techie, you can skip this info.

This icon denotes the programs that are included on this book's CD-ROM.

Where to Go from Here

You can find a set of errata and Frequently Asked Questions for this and all my books at `www.stephendavis.com`. You will also find a link to my e-mail address there. Feel free to send me your questions and comments (that's how I learn). It's through reader input that these books can improve.

Now you've stalled long enough, it's time to turn to Chapter 1 and start discovering how to program!

Part I
Let's Get Started

The 5th Wave By Rich Tennant

"We're outsourcing everything but our core competency. Once we find out what that is, we'll begin the outsourcing process."

In this part . . .

You will learn what it means to program a computer. You will also get your first taste of programming — I take you through the steps to enter, build, and execute your first program. It will all be a bit mysterious in this part, but things will clear up soon, I promise.

Chapter 1

What Is a Program?

In This Chapter

- Understanding programs
- Writing your first "program"
- Looking at computer languages

In this chapter, you will learn what a program is and what it means to write a program. You'll practice on a Human Computer. You'll then see some program snippets written for a real computer. Finally, you'll see your first code snippet written in C++.

Up until now all of the programs running on your computer were written by someone else. Very soon now, that won't be true anymore. You will be joining the ranks of the few, the proud: the programmers.

How Does My Son Differ from a Computer?

A computer is an amazingly fast but incredibly stupid machine. A computer can do anything you tell it (within reason), but it does *exactly* what it's told — nothing more and nothing less.

In this respect, a computer is almost the exact opposite of a human: humans respond intuitively. When I was learning a second language, I learned that it wasn't enough to understand what was being said — it's just as important and considerably more difficult to understand what was left unsaid. This is information that the speaker shares with the listener through common experience or education — things that don't need to be said.

For example, I say things to my son like, "Wash the dishes" (for all the good it does me). This seems like clear enough instructions, but the vast majority of the information contained in that sentence is implied and unspoken.

Let's assume that my son knows what dishes are and that dirty dishes are normally in the sink. But what about knives and forks? After all, I only said dishes, I didn't say anything about eating utensils, and don't even get me started on glassware. And did I mean wash them manually, or is it okay to load them up into the dishwasher to be washed, rinsed, and dried automatically?

But the fact is, "Wash the dishes" is sufficient instruction for my son. He can decompose that sentence and combine it with information that we both share, including an extensive working knowledge of dirty dishes, to come up with a meaningful understanding of what I want him to do — whether he does it or not is a different story. I would guess that he can perform all the mental gymnastics necessary to understand that sentence in about the same amount of time that it takes me to say it — about 1 to 2 seconds.

A computer can't make heads or tails out of something as vague as "wash the dishes." You have to tell the computer exactly what to do with each different type of dish, how to wash a fork, versus a spoon, versus a cup. When does the program stop washing a dish (that is, how does it know when a dish is clean)? When does it stop washing (that is, how does it know when it's finished)?

My son has gobs of memory — it isn't clear exactly how much memory a normal human has, but it's boat loads. Unfortunately, human memory is fuzzy. For example, witnesses to crimes are notoriously bad at recalling details even a short time after the event. Two witnesses to the same event often disagree radically on what transpired.

Computers also have gobs of memory, and that's very good. Once stored, a computer can retrieve a fact as often as you like without change. As expensive as memory was back in the early 1980s, the original IBM PC had only 16K (that's 16 thousand bytes). This could be expanded to a whopping 64K. Compare this with the 1GB to 3GB of main storage available in most computers today (1GB is *one billion bytes*).

As expensive as memory was, however, the IBM PC included extra memory chips and decoding hardware to detect a memory failure. If a memory chip went bad, this circuitry was sure to find it and report it before the program went haywire. This so-called Parity Memory was no longer offered after only a few years, and as far as I know, it is unavailable today except in specific applications where extreme reliability is required — because the memory boards almost never fail.

On the other hand, humans are very good at certain types of processing that computers do poorly, if at all. For example, humans are very good at pulling the meaning out of a sentence garbled by large amounts of background noise. By contrast, digital cell phones have the infuriating habit of just going silent whenever the noise level gets above a built-in threshold.

Programming a "Human Computer"

Before I dive into showing you how to write programs for computer consumption, I start by showing you a program to guide human behavior so that you can better see what you're up against. Writing a program to guide a human is much easier than writing programs for computer hardware because we have a lot of familiarity with and understanding of humans and how they work (I assume). We also share a common human language to start with. But to make things fair, assume that the human computer has been instructed to be particularly literal — so the program will have to be very specific. Our guinea pig computer intends to take each instruction quite literally.

The problem I have chosen is to instruct our human computer in the changing of a flat tire.

The algorithm

The instructions for changing a flat tire are straightforward and go something like the following:

1. Raise the car.
2. Remove the lug nuts that affix the faulty tire to the car.
3. Remove the tire.
4. Mount the new tire.
5. Install the lug nuts.
6. Lower the car.

(I know that technically the lug nuts hold the wheel onto the car and not the tire, but that distinction isn't important here. I use the terms "wheel" and "tire" synonymously in this discussion.)

As detailed as these instructions might seem to be, this is not a program. This is called an *algorithm*. An algorithm is a description of the steps to be performed, usually at a high level of abstraction. An algorithm is detailed but general. I could use this algorithm to repair any of the flat tires that I have experienced or ever will experience. But an algorithm does not contain sufficient detail for even our intentionally obtuse human computer to perform the task.

The Tire Changing Language

Before we can write a program, we need a language that we can all agree on. For the remainder of this book, that language will be C++, but I use the newly invented TCL (Tire Changing Language) for this example. I have specifically adapted TCL to the problem of changing tires.

TCL includes a few nouns common in the tire-changing world:

- car
- tire
- nut
- jack
- toolbox
- spare tire
- wrench

TCL also includes the following verbs:

- grab
- move
- release
- turn

Finally, the TCL-executing processor will need the ability to count and to make simple decisions.

This is all that the tire-changing robot understands. Any other command that's not part of Tire Changing Language generates a blank stare of incomprehension from the human tire-changing processor.

The program

Now it's time to convert the algorithm, written in everyday English, into a program written in Tire Changing Language. Take the phrase, "Remove the lug nuts." Actually, quite a bit is left unstated in that sentence. The word "remove" is not in the processor's vocabulary. In addition, no mention is made of the wrench at all.

The following steps implement the phrase "Remove a lug nut" using only the verbs and nouns contained in Tire Changing Language:

1. Grab wrench.
2. Move wrench to lug nut.
3. Turn wrench counterclockwise five times.
4. Move wrench to toolbox.
5. Release wrench.

I didn't explain the syntax of Tire Changing Language. For example, the fact that every command starts with a single verb or that the verb "grab" requires a single noun as its object and that "turn" requires a noun, a direction, and a count of the number of turns to make. Even so, the program snippet should be easy enough to read (remember that this is not a book about Tire Changing Language).

You can skate by on Tire Changing Language, but you will have to learn the grammar of each C++ command.

The program begins at Step 1 and continues through each step in turn until reaching Step 5. In programming terminology, we say that the program flows from Step 1 through Step 5. Of course, the program's not going anywhere — the processor is doing all the work, but the term "program flow" is a common convention.

Even a cursory examination of this program reveals a problem: What if there is no lug nut? I suppose it's harmless to spin the wrench around a bolt with no nut on it, but doing so wastes time and isn't my idea of a good solution. The Tire Changing Language needs a branching capability that allows the program to take one path or another depending upon external conditions. We need an IF statement like the following:

1. Grab wrench.
2. If lug nut is present

3. {
4. Move wrench to lug nut.
5. Turn wrench counterclockwise five times.
6. }
7. Move wrench to toolbox.
8. Release wrench.

The program starts with Step 1 just as before and grabs a wrench. In the second step, however, before the program waves the wrench uselessly around an empty bolt, it checks to see if a lug nut is present. If so, flow continues on with Steps 3, 4, and 5 as before. If not, however, program flow skips these unnecessary steps and goes straight on to Step 7 to return the wrench to the toolbox.

In computerese, you say that the program executes the logical expression "is lug nut present?" This expression returns either a true (yes, the lug nut is present) or a false (no, there is no lug nut there).

What I call steps, a programming language would normally call a *statement*. An *expression* is a type of statement that returns a value, such as 1 + 2 is an expression. A *logical expression* is an expression that returns a true or false value, such as "is the author of this book handsome?" is `true`.

The braces in Tire Changing Language are necessary to tell the program which steps are to be skipped if the condition is not true. Steps 4 and 5 are executed only if the condition is true.

I realize that there's no need to grab a wrench if there's no lug to remove, but work with me here.

This improved program still has a problem: How do you know that 5 turns of the wrench will be sufficient to remove the lug nut? It most certainly will not be for most of the tires with which I am familiar. You could increase the number of turns to something that seems likely to be more than enough, say 25 turns. If the lug nut comes loose after the twentieth turn, for example, the wrench will turn an extra 5 times. This is a harmless but wasteful solution.

A better approach is to add some type of "loop and test" statement to the Tire Changing Language:

1. Grab wrench.
2. If lug nut is present

```
3. {
4.     Move wrench to lug nut.
5.     While (lug nut attached to car)
6.     {
7.         Turn wrench counterclockwise one turn.
8.     }
9. }
10. Move wrench to toolbox.
11. Release wrench.
```

Here the program flows from Step 1 through Step 4 just as before. In Step 5, however, the processor must make a decision: Is the lug nut attached? On the first pass, we will assume that the answer is yes so that the processor will execute Step 7 and turn the wrench counterclockwise one turn. At this point, the program returns to Step 5 and repeats the test. If the lug nut is still attached, the processor repeats Step 7 before returning to Step 5 again. Eventually, the lug nut will come loose and the condition in Step 5 will return a false. At this point, control within the program will pass on to Step 9, and the program will continue as before.

This solution is superior to its predecessor: It makes no assumptions about the number of turns required to remove a lug nut. It is not wasteful by requiring the processor to turn a lug nut that is no longer attached, nor does it fail because the lug nut is only half removed.

As nice as this solution is, however, it still has a problem: It removes only a single lug nut. Most medium-sized cars have five nuts on each wheel. We could repeat Steps 2 through 9 five times, once for each lug nut. However, this doesn't work very well either. Most compact cars have only four lug nuts, and large pickups have up to eight.

The following program expands our grammar to include the ability to loop across lug nuts. This program works irrespective of the number of lug nuts on the wheel:

```
1. Grab wrench.
2. For each lug bolt on wheel
3. {
4.     If lug nut is present
5.     {
```

```
6.          Move wrench to lug nut.
7.          While (lug nut attached to car)
8.          {
9.              Turn wrench counterclockwise one turn.
10.         }
11.     }
12. }
13. Move wrench to toolbox.
14. Release wrench.
```

This program begins just as before with the grabbing of a wrench from the toolbox. Beginning with Step 2, however, the program loops through Step 12 for each lug nut bolt on the wheel.

Notice how Steps 7 through 10 are still repeated for each wheel. This is known as a *nested* loop. Steps 7 through 10 are called the inner loop, while Steps 2 through 12 are the outer loop.

The complete program consists of the addition of similar implementations of each of the steps in the algorithm.

Computer processors

Removing the wheel from a car seems like such a simple task, and yet it takes 11 instructions in a language designed specifically for tire changing just to get the lug nuts off. Once completed, this program is likely to include over 60 or 70 steps with numerous loops. Even more if you add in logic to check for error conditions like stripped or missing lug nuts.

Think of how many instructions have to be executed just to do something as mundane as move a window about on the display screen (remember that a typical screen may have 1280 x 1024 or a little over a million pixels or more displayed). Fortunately, though stupid, a computer processor is very fast. For example, the processor that's in your PC can likely execute several billion instructions per second. The instructions in your generic processor don't do very much — it takes several instructions just to move one pixel — but when you can rip through a billion or so at a time, scrolling a mere million pixels becomes child's play.

The computer will not do anything that it hasn't already been programmed for. The creation of a Tire Changing Language was not enough to replace my flat tire — someone had to write the program instructions to map out step by step what the computer will do. And writing a real-world program designed to handle all of the special conditions that can arise is not an easy task. Writing an industrial-strength program is probably the most challenging enterprise you can undertake.

So the question becomes: "Why bother?" Because once the computer is properly programmed, it can perform the required function repeatedly, tirelessly, and usually at a greater rate than is possible under human control.

Computer Languages

The Tire Changing Language isn't a real computer language, of course. Real computers don't have machine instructions like "grab" or "turn." Worse yet, computers "think" using a series of ones and zeros. Each internal command is nothing more than a sequence of binary numbers. Real computers have instructions like 01011101, which might add 1 to a number contained in a special purpose register. As difficult as programming in TCL might be, programming by writing long strings of numbers is even harder.

The native language of the computer is known as *machine language* and is usually represented as a sequence of numbers written either in binary (base 2) or hexadecimal (base 16). The following represents the first 64 bytes from the Conversion program in Chapter 3.

```
<main+0>: 01010101 10001001 11100101 10000011 11100100 11110000 10000011 11101100
<main+8>: 00100000 11101000 00011010 01000000 00000000 00000000 11000111 01000100
<main+16>:00100100 00000100 00100100 01110000 01000111 00000000 11000111 00000100
<main+24>:00100100 10000000 01011111 01000111 00000000 11101000 10100110 10001100
<main+32>:00000110 00000000 10001101 01000100 00100100 00010100 10001001 01000100
```

Fortunately, no one writes programs in machine language anymore. Very early on, someone figured out that it is much easier for a human to understand ADD 1,REG1 as "add 1 to the value contained in register 1," rather than 01011101. In the "post-machine language era," the programmer wrote her programs in this so-called *assembly language* and then submitted it to a program called an *assembler* that converted each of these instructions into their machine-language equivalent.

The programs that people write are known as *source code* because they are the source of all evil. The ones and zeros that the computer actually executes are called *object code* because they are the object of so much frustration.

The following represents the first few assembler instructions from the `Conversion` program when compiled to run on an Intel processor executing Windows. This is the same information previously shown in binary form.

```
<main>:         push   %ebp
<main+1>:       mov    %esp,%ebp
<main+3>:       and    $0xfffffff0,%esp
<main+6>:       sub    $0x20,%esp
<main+9>:       call   0x40530c <__main>
<main+14>:      movl   $0x477024,0x4(%esp)
<main+22>:      movl   $0x475f80,(%esp)
<main+29>:      call   0x469fac <operator<<>
<main+34>:      lea    0x14(%esp),%eax
<main+38>:      mov    %eax,0x4(%esp)
```

This is still not very intelligible, but it's clearly a lot better than just a bunch of ones and zeros. Don't worry — you won't have to write any assembly language code in this book either.

The computer does not actually ever execute the assembly language instructions. It executes the machine instructions that result from converting the assembly instructions.

High level languages

Assembly language might be easier to remember, but there's still a lot of distance between an algorithm like the tire-changing algorithm and a sequence of MOVEs and ADDs. In the 1950s, people started devising progressively more expressive languages that could be automatically converted into machine language by a program called a *compiler.* These were called *high level languages* because they were written at a higher level of abstraction than assembly language.

One of the first of these languages was COBOL (Common Business Oriented Language). The idea behind COBOL was to allow the programmer to write commands that were as much like English sentences as possible. Suddenly programmers were writing sentences like the following to convert temperature from Celsius to Fahrenheit (believe it or not, this is exactly what the machine and assembly language snippets shown earlier do):

```
INPUT CELSIUS_TEMP
SET FAHRENHEIT_TEMP TO CELSIUS_TEMP * 9/5 + 32
WRITE FAHRENHEIT_TEMP
```

The first line of this program reads a number from the keyboard or a file and stores it into the variable `CELSIUS_TEMP`. The next line multiplies this number by ⅘ and adds 32 to the result to calculate the equivalent temperature in Fahrenheit. The program stores this result into a variable called `FAHRENHEIT_TEMP`. The last line of the program writes this converted value to the display.

People continued to create different programming languages, each with its own strengths and weaknesses. Some languages, like COBOL, were very wordy but easy to read. Other languages were designed for very specific areas like database languages or the languages used to create interactive Web pages. These languages include powerful constructs designed for one specific problem area.

The C++ language

C++ (pronounced "C plus plus," by the way) is a symbolically oriented high level language. C++ started out life as simply C in the 1970s at Bell Labs. A couple of guys were working on a new idea for an operating system known as Unix (the predecessor to Linux and Mac OS and still used across industry and academia today). The original C language created at Bell Labs was modified slightly and adopted as a worldwide ISO standard in early 1980. C++ was created as an extension to the basic C language mostly by adding the features that I discuss in Parts V and VI of this book.When I say that C++ is symbolic, I mean that it isn't very wordy, preferring to use symbols rather than long words like in COBOL. However, C++ is easy to read once you are accustomed to what all the symbols mean. The same Celsius to Fahrenheit conversion code shown in COBOL earlier appears as follows in C++:

```
cin >> celsiusTemp;
fahrenheitTemp = celsiusTemp * 9 / 5 + 32;
cout << fahrenheitTemp;
```

The first line reads a value into the variable `celsiusTemp`. The subsequent calculation converts this Celsius temperature to Fahrenheit like before, and the third line outputs the result.

C++ has several other advantages compared with other high level languages. For one, C++ is universal. There is a C++ compiler for almost every computer in existence.

In addition, C++ is efficient. The more things a high level language tries to do automatically to make your programming job easier, the less efficient the machine code generated tends to be. That doesn't make much of a difference for a small program like most of those in this book, but it can make a big difference when manipulating large amounts of data, like moving pixels around on the screen, or when you want blazing real-time performance. It's not an accident that Unix and Windows are written in C++ and the Macintosh O/S is written in a language very similar to C++.

Chapter 2

Installing Code::Blocks

In This Chapter

- Reviewing the compilation process
- Installing the Code::Blocks development environment
- Testing your installation with a default program
- Reviewing the common installation errors

In this chapter, you will review what it takes to create executable programs from C++ source code that you can run on the Windows, Linux, or Macintosh computer. You will then install the Code::Blocks integrated development environment used in the remainder of the book, and you will build a default test program to check out your installation. If all is working, by the time you reach the end of this chapter, you will be ready to start writing and building C++ programs of your own — with a little help, of course!

Reviewing the Compilation Process

You need two programs to create your own C++ programs. First, you need a text editor that you can use to enter your C++ instructions. Any editor capable of generating straight ASCII text letters will work. I have written programs using the Notepad editor that comes with Windows. However, an editor that knows something about the syntax of C++ is preferable since it can save you a lot of typing and sometimes highlight mistakes that you might be making as you type, in much the same way that a spelling checker highlights misspelled words in a word processor.

The second program you will need is a compiler that converts your C++ source statements into machine language that the computer can understand and interpret. This process of converting from source C++ statements to object machine code is called *building*. Graphically, the process looks something like that shown in Figure 2-1.

The process of building a program actually has two steps: The C++ compiler first converts your C++ source code statements into a machine executable format in a step known as *compiling*. It then combines the machine instructions from your program with instructions from a set of libraries that come standard with C++ in a second step known as *linking* to create a complete executable program.

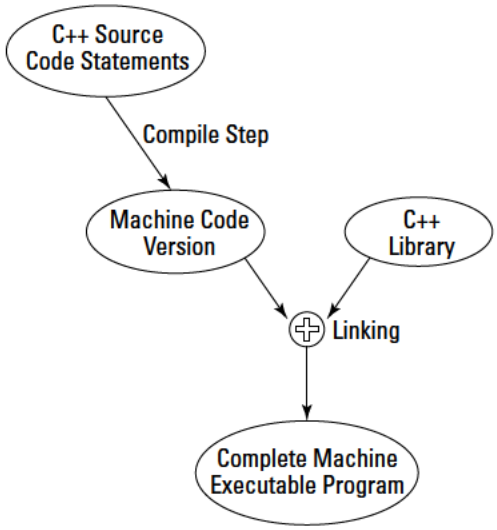


Figure 2-1: The C++ program development process.

Most C++ compilers these days come in what is known as an Integrated Development Environment or IDE. These IDEs include the editor, the compiler, and several other useful development programs together in a common package. Not only does this save you from the need to purchase these programs separately, but combining them into a single package produces several productivity benefits. First, the editor can invoke the compiler quickly without the need for you to switch back and forth manually. In addition, the editors in most IDEs provide quick and efficient means for finding and fixing coding errors.

Some IDEs include visual programming tools that allow the programmer to draw common windows such as dialog boxes on the display — the IDE generates the C++ code necessary to display these boxes automatically.

As nice as that sounds, the automatically generated code only displays the windows. A programmer still has to generate the real code that gets executed whenever the operator selects buttons within those windows.

Invariably, these visual IDEs are tightly coupled into one or the other operating system. For example, the popular Visual Studio is strongly tied into the

.NET environment in Windows. It is not possible to use Visual Studio without learning the .NET environment and something about Windows along with C++ (or one of the other .NET languages). In addition, the resulting programs only run in a .NET environment.

In this book, you will use a public domain C++ IDE known as Code::Blocks. Versions of Code::Blocks exist for Windows, Linux, and MacOS — a version for Windows is included on the CD-ROM accompanying this book. Versions of Code::Blocks for Macintosh and Linux are available for free download at `www.codeblocks.org`.

You will use Code::Blocks to generate the programs in this book. These programs are known as *Console Applications* since they take input from and display text back to a console window. While this isn't as sexy as windowed development, staying with Console Applications will allow you to focus on C++ and not be distracted by the requirements of a windowed environment. In addition, using Console Applications will allow the programs in the book to run the same on all environments that are supported by Code::Blocks.

Installing Code::Blocks

Beginning Programming with C++ For Dummies includes a version of Code::Blocks for Windows on the CD-ROM. This section provides detailed installation instructions for this version. The steps necessary to download and install versions of Code::Blocks from `www.codeblocks.org` will be similar.

1. **Insert the enclosed CD-ROM into your computer.**

   That's straightforward enough.

2. **Read the End User License Agreement (EULA) and select Accept.**

3. **Select the Software tab and then select Code::Blocks to install the Code::Blocks environment.**

   On some versions of Windows, you may see a message appear that "An unidentified program wants access to your computer." Of course, that unidentified program is the Code::Blocks Setup program.

4. **Select Allow.**

   Setup now unpacks the files it needs to start and run the Code::Blocks Setup Wizard. This may take about a minute. Once it finishes, the startup window shown in Figure 2-2 appears.

5. **Close any other programs that you may be executing and select Next.**

   The Setup Wizard displays the generic End User License Agreement (EULA). There's nothing much here to get excited about.

Figure 2-2: The Code::Blocks Setup Wizard guides you through the installation process.

6. **Select I Agree.**

   The Setup Wizard then displays a list of the components that you may choose to install. The defaults are okay, but you may want to also check the Desktop Shortcut option as shown in Figure 2-3. Doing this provides an icon on the desktop that you can use to start Code::Blocks without going through the Program Files menu.

Figure 2-3: Checking Desktop Shortcut creates an icon that you can use to start Code::Blocks more quickly.

7. **Select Next.**

   The next window asks you to choose the install location. This window also tells you how much hard disk space Code::Blocks requires (about 150MB, depending upon the options you've selected) and how much space is available on your hard drive. If you don't have enough free disk space, you'll have to delete some of those YouTube videos you've captured to make room before you continue.

8. **The default install location is fine, so once you have enough disk space, select Install.**

   At this point, the Code::Blocks Setup Wizard really goes to work. It extracts umpteen dozen files that it installs in a myriad of subdirectories too complicated for mere mortals. This process may take several minutes.

9. **When the installation is complete, a dialog box appears asking you whether you want to run Code::Blocks now. Select No.**

   If all has gone well so far, the Installation Complete window shown in Figure 2-4 appears.

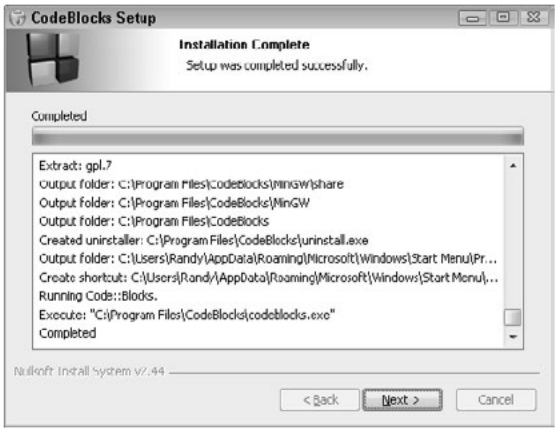


Figure 2-4: The Installation Complete window signals that Code::Blocks has been successfully installed.

10. **Click Next.**

    Finally, the Completing the Code::Blocks Setup Wizard window appears. This final step creates the icons necessary to start the application.

11. **Click Finish.**

You've done it! You've installed Code::Blocks. All that remains now is to test whether it works, and then you'll be ready to start programming.

Testing the Code::Blocks Installation

In this section, you will build a default program that comes with Code::Blocks. This program does nothing more than display "Hello, world!" on the display, but building and running this program successfully will verify that you've installed Code::Blocks properly.

1. **Start Code::Blocks by double-clicking on the Code::Blocks icon created on the Desktop or selecting Programs⇨Code::Blocks⇨Code::Blocks.**

   This should open a window like the one shown in Figure 2-5.

Across the top of the window are the usual menu options starting with File, Edit, View, and so on. The window at the upper right, the one that says "Start here," is where the source code will go when you get that far. The window at the lower right is where Code::Blocks displays messages to the user. Compiler error messages appear in this space. The window on the left labeled Management is where Code::Blocks keeps track of the files that make up the programs. It should be empty now since you have yet to create a program. The first thing you will need to do is create a project.

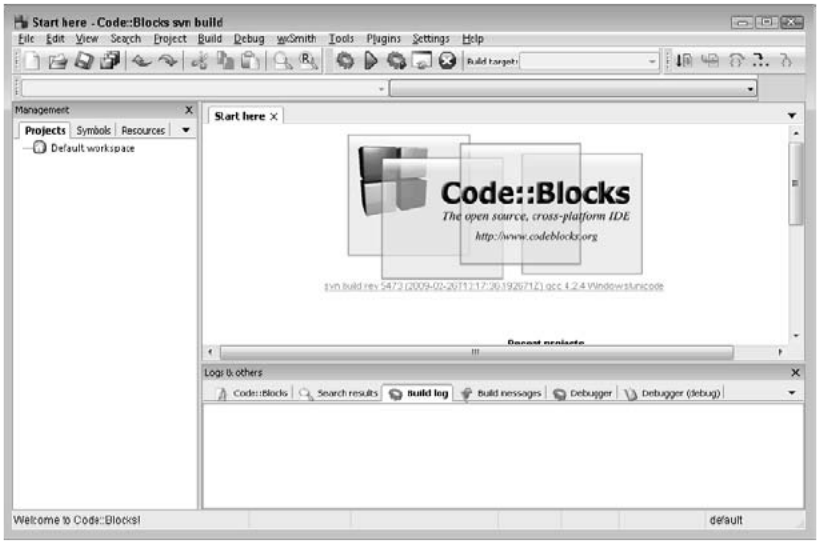


Figure 2-5: The opening screen of the Code::Blocks environment.

What's a project?

You want Code::Blocks to create only Console Applications, but it can create a lot of different types of programs. For Windows programmers, Code::Blocks can create Dynamic Link Libraries (also known simply as DLLs). It can create Windows applications. It can create both static and dynamically linked libraries for Linux and MacOS.

In addition, Code::Blocks allows the programmer to set different options on the ways each of these targets is built. I will show you how to adjust a few of these settings in later chapters. And finally, Code::Blocks remembers how you have your windows configured for each project. When you return to the project, Code::Blocks restores the windows to their last configuration to save you time.

Code::Blocks stores the information it needs about the type of program that you are building, the optional settings, and the window layout in two project files. The settings are stored in a file with the same name as the program but carrying the extension `.cbp`. The window configuration is stored in a file with the same name but with the extension `.layout`.

Creating the project

1. **Select File➪New➪Projects to open the window shown in Figure 2-6.**

   This is a list of all of the types of applications that Code::Blocks knows how to build.

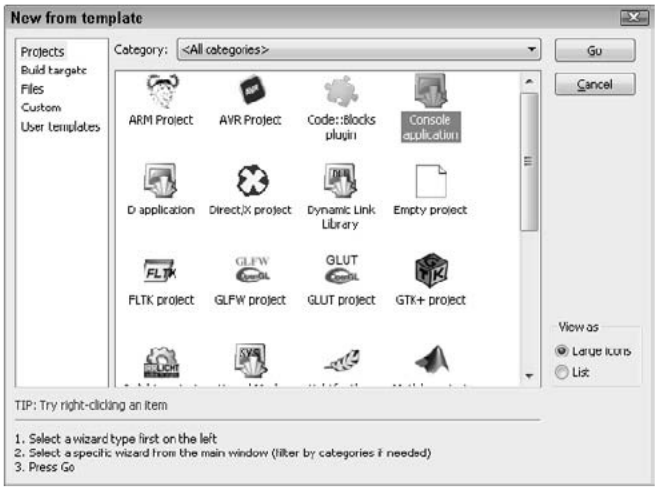


Figure 2-6: Select the Console Application from the many types of targets offered.

   Fortunately, you will be concentrating on just one, the Console Application.

2. **Select Console Application and select Go.**

   Code::Blocks responds with the display shown in Figure 2-7. Here Code::Blocks is offering you the option to create either a C or a C++ program.

3. **Select C++ and click Next.**

   Code::Blocks opens a dialog box where you will enter the name and optional subfolder for your project. First, click on the little ... button to create a folder to hold your Projects, navigate to the root of your working disk (on a Windows machine, it'll be either C or D, most likely C). Select the Make New Folder button at the bottom left of the window. Name the new folder `Beginning_Programming-CPP`.

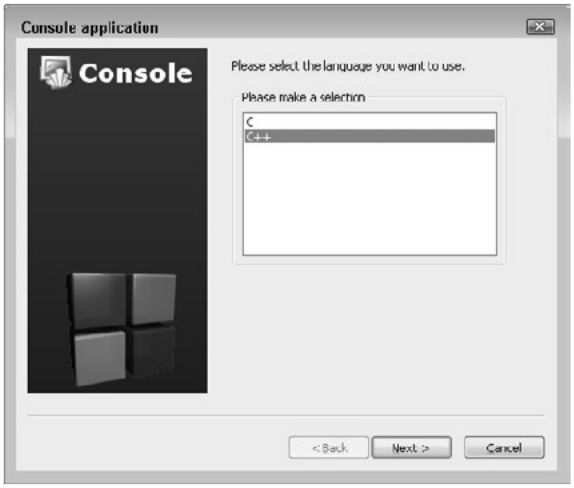


Figure 2-7: Select C++ as your language of choice.

4. **Click OK when your display looks like the one shown in Figure 2-8.**

The folder that you create to hold your project must not contain any spaces in the name. In addition, none of the folders in the path should contain spaces. That automatically eliminates placing your projects on the Desktop since the path to the Desktop contains spaces. You should also avoid spaces in the name of the Project. You can use underscores to separate words instead. The Code::Blocks compiler gets confused with spaces in the filenames and generates obscure and largely meaningless errors.

Figure 2-8: Create the folder `Beginning_Programming-CPP` into which you will collect your C++ projects.

Now enter the name of the Project as **HelloWorld**. Notice that Code::Blocks automatically creates a subfolder of the same name to contain the files that make up the project.

5. **Click Next when your display looks like Figure 2-9.**

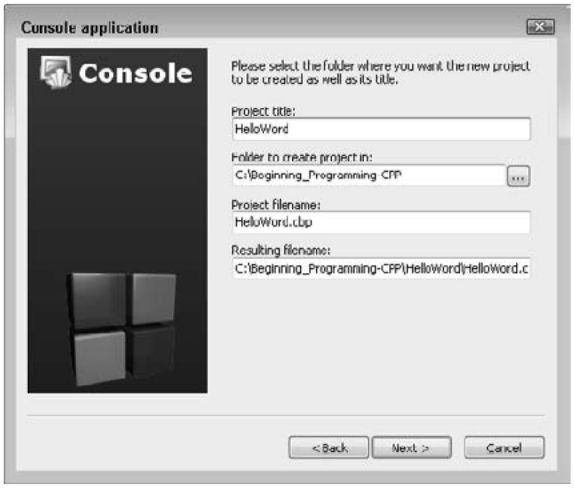


Figure 2-9: Call your first project HelloWorld.

6. **When Code::Blocks asks how you want your subfolders set up, you can accept the default configuration, as shown in Figure 2-10. Select Finish.**

You can select the Back button to back up to a previous menu in the preceding steps if you screw something up. However, you may have to reenter any data you entered when you go forward again. Once you select Finish, you can no longer return and change your selections. If you screw up and want to redo the project, you will first need to remove the Project: Right-click on HelloWorld in the Management window and select Close Project. Now you can delete the folder `Beginning_Programming-CPP\HelloWorld` and start over again.

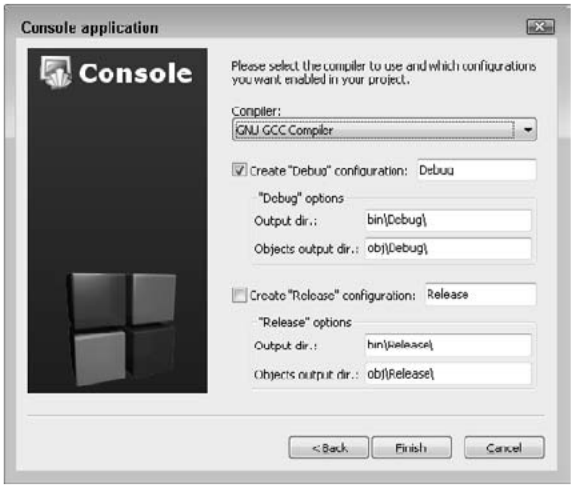


Figure 2-10: Select Finish on the final page to complete the creation of the HelloWorld Project.

Testing your default project

Code::Blocks creates a Console Application project and even populates it with a working program when you select Finish on the Project Wizard. To see that program, click on the plus (+) sign next to Sources in the Management window on the left side of the display. The drop-down list reveals one file, `main.cpp`. Double-click on `main.cpp` to display the following simple program in the source code entry window on the right:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

I'll skip over what some of this stuff means for now, but the crux of the program starts after the open brace following `main()`. This is where execution begins. The line

```
cout << "Hello world!" << endl;
```

says output the line "Hello, world!" to the `cout`, which by default is the command line. The next line

```
return 0;
```

causes control to return to the operating system, which effectively terminates the program.

1. **Select Build⇨Build to build the C++ source statements into an executable machine language program.**

   (You can press Ctrl+F9 or click the Build icon if you prefer.) Immediately, you should see the Build Log tab appear in the lower-right screen followed by a series of lengthy commands, as shown in Figure 2-11. This is Code::Blocks telling the C++ compiler how to build the test program using the settings stored in the project file. The details aren't important. What is important, however, is that the final two lines of the Build Log window should be

   ```
   Process terminated with status 0 (0 minutes, 1 seconds)
   0 errors, 0 warnings
   ```

The terminated status of 0 means that the build process worked properly. The "0 errors, 0 warnings" means that the program compiled without errors or warnings. (The build time of 1 second is not important.)

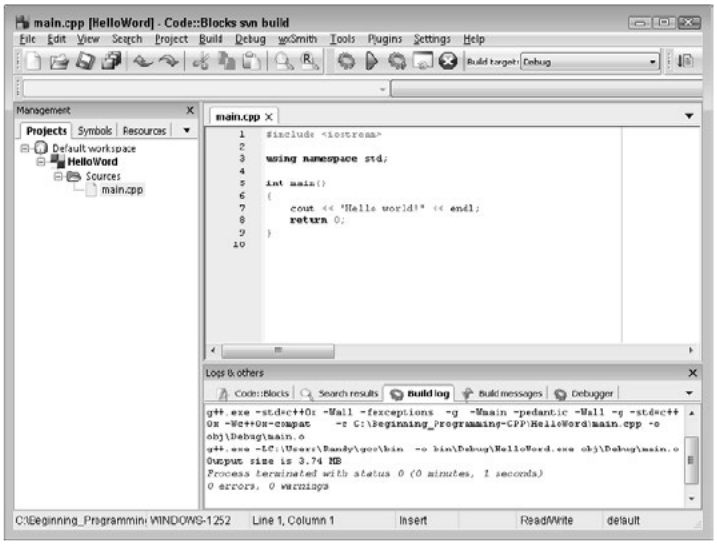


Figure 2-11: Building the default program should result in a working program with no errors and no warnings.

If you don't get a status of 0 with 0 errors and 0 warnings, then something is wrong with your installation or with the project. The most common sources of error are

- You already had a gcc compiler installed on your computer before you installed Code::Blocks. Code::Blocks uses a special version of the GNU gcc compiler, but it will use any other gcc compiler that you may already have installed on your computer. Your safest bet is to uninstall Code::Blocks, uninstall your other gcc compiler, and reinstall Code::Blocks from scratch.
- You built your project in a directory that contains a space in the name; for example, you built your project on the Desktop. Be sure to build your project in the folder `Beginning_Programming-CPP` in the root of your user disk (most likely C on a Windows machine).
- You built a project directly from the enclosed CD-ROM that came with the book. (This doesn't apply to the steps here, but it is a common source of error anyway. You can't build your program on a read-only storage medium like a CD-ROM. You will have to copy the files from the CD-ROM to the hard drive first.)

2. **Select Build⇨Run (Ctrl+F10) to execute the program.**

   Immediately a window should pop open with the message "Hello, world!" followed by the return code of zero and the message "Press any key to continue," as shown in Figure 2-12.

3. **Press Enter**.

   The window will disappear and control returns to the Code::Blocks text editor.

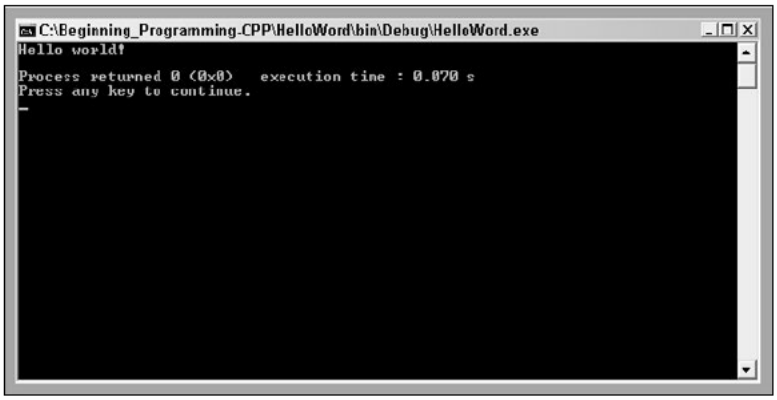


Figure 2-12: The default program displays "Hello, world!" and waits for you to press a key.

If you were able to see the "Hello, world!" message by executing the program, then congratulations! You've installed your development environment and built and executed your first C++ program successfully. If you did not, then delete the `Beginning_Programming_CPP` folder, uninstall Code::Blocks, and try again, carefully comparing your display to the figures shown in this chapter. If you are still having problems, refer to `www.stephendavis.com` for pointers as to what might be wrong, as well as a link to my e-mail where you can send me questions and comments. I cannot do your programming homework for you, but I can answer questions to get you started.

Chapter 3

Writing Your First Program

In This Chapter

- Entering your first C++ program
- Compiling and executing your program
- Examining some things that could go wrong
- Executing your program
- Reviewing how the program works

This chapter will guide you through the creation of your first program in C++. You will be using the Code::Blocks C++ environment. It will all be a bit "cookbookish" since this is your first time. I explain all of the parts that make up this program in subsequent chapters beginning with Part II, but for now you'll be asked to accept a few things on faith. Soon all will be revealed, and everything you do in this chapter will make perfect sense.

Creating a New Project

As always, you must create a new project to house your program. Follow the abbreviated steps here (or you can use the detailed steps from Chapter 2):

1. **With Code::Blocks open, select File➪New➪Project.**
2. **Select Console Applications and select Go (or double-click on the Console Applications icon).**
3. **Select C++ as your language of choice and select Next.**
4. **Enter** Conversion **as the Project Title.**

   If you followed the steps in Chapter 2, the "Folder to create project in" should already be set to `Beginning_Programming-CPP`. If not, it's not too late to click the ... button and create the folder in the root of your working disk. (This is described in detail in Chapter 2.) The Code::Blocks Wizard fills in the name of the Project and the name of the resulting program for you.

When you're done, your window should look like that shown in Figure 3-1.

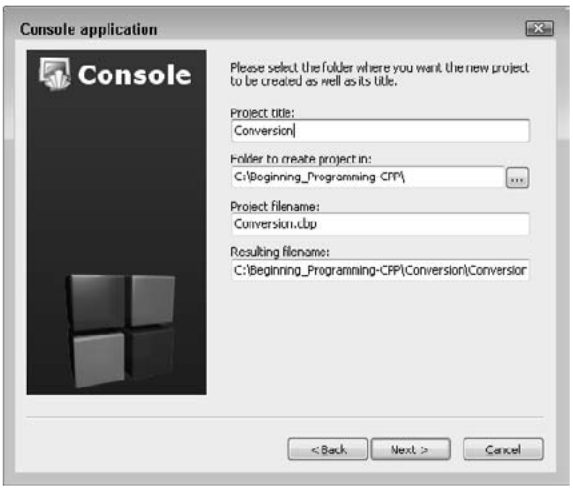


Figure 3-1: The Project window for the Conversion program.

5. **Select Next.**

   The next window allows you to change the target folders. The defaults are fine.

6. **Select Finish.**

Code::Blocks creates a new Project and adds it to the earlier `HelloWorld` project. (See the "Organizing projects" sidebar for an explanation of why this happens.) The resulting display should look like Figure 3-2.

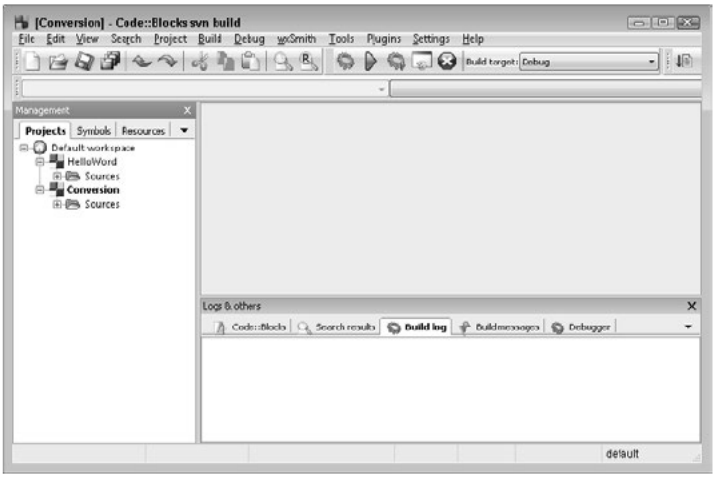


Figure 3-2: The initial display after creating the Conversion project.

Organizing projects

You may be curious as to why Code::Blocks added the new Conversion project to the existing HelloWorld project rather than replacing it. A large effort involving multiple developers may be broken up into a number of different programs that are all designed to work together. To support this, Code::Blocks allows you to have any number of different projects loaded at once.

The collection of all projects is called a *workspace*. Since you didn't specify a workspace when you started Code::Blocks, the projects you've created so far have been going into the *default workspace*. Only one project in the workspace can be active at a time. This is the project that appears in bold (refer to Figure 3-2 again, and you'll notice that `Conversion` is bolded while `HelloWorld` is not). Any Code::Blocks commands you perform are directed at the active project. By default, the last project you created is the active project, but you can change the active project by right-clicking on it in the Management window and selecting Activate Project (the first option in the list).

If you were to take a peek in the `Beginning_Programming-CPP` folder right now, you would notice two subfolders: `HelloWorld` and `Conversion`. Both of these subfolders include a project file with the extension .cbp that contains the compiler settings, a layout file with the extension layout that describes the way you want your windows set up when working on this project, and the file `main.cpp` that contains the C++ program created by the application wizard. `HelloWorld` contains a further subfolder named `Debug`.

C++ programs can have any name that you like, but it should end in .cpp. You will see how to create multiple C++ source files with different names in Chapter 12.

Entering Your Program

It is now time to enter your first program using the following steps:

1. **Make sure that Conversion is bolded in the Management window (refer to Figure 3-2).**

   This indicates that it's the active project. If it is not, right-click on Conversion and select Activate Project from the drop-down menu.

2. **Close any source file windows that may be open by selecting File➪Close all files.**

   Alternatively, you can close just the source files you want by clicking on the small X next to the name of the file in the editor tab. You don't want to inadvertently edit the wrong source file.

3. **Open the Sources folder by clicking on the small plus sign next to Sources underneath Conversion in the Management window.**

   The drop-down menu should reveal the single file `main.cpp`.

Filename extensions

Windows has a bad habit of hiding the filename extensions when displaying filenames. For some applications this may be a good idea, but this is almost never a good idea for a programmer. With extensions hidden, Windows may display three or four files with the same name `HelloWorld`. This confusing state of affairs is easily cleared up when you display file extensions and realize that they are all different.

You should disable the Windows Hide Extensions feature. Exactly how you do this depends upon what version of Windows you are using:

- Windows 2000: Select Start➪Settings➪Control Panel➪Folder Options.
- Windows XP with Default View: Select Start➪Control Panel➪Performance and Maintenance➪File Types.
- Windows XP with Classic view: Select Start➪Control Panel➪Folder options.
- Windows Vista with Default view: Select Start➪Control Panel➪Appearance and Personalization➪Show hidden files and folders.
- Windows Vista with Classic view: Select Start➪Settings➪Control Panel➪Folder options.

Now navigate to the View tab of the Folder Options dialog box that appears. Scroll down until you find "Hide extensions for known file types." Make sure that this box is unchecked. Select OK to close the dialog box.

4. **Double-click `main.cpp` to open the file in the editor.**
5. **Edit the contents of `main.cpp` by entering the following program exactly as it appears here.**

The result is shown in Figure 3-3.

This is definitely the hard part, so take your time and be patient:


```
//
//  Conversion - Program to convert temperature from
//             Celsius degrees into Fahrenheit:
//             Fahrenheit = Celsius  * (212 - 32)/100 + 32
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
  // enter the temperature in Celsius
  int celsius;
  cout << "Enter the temperature in Celsius:";
  cin >> celsius;

  // convert Celsius into Fahrenheit values
  int fahrenheit;
```

```
  fahrenheit = celsius * 9/5 + 32;

  // output the results (followed by a NewLine)
  cout << "Fahrenheit value is:";
  cout << fahrenheit << endl;

  // wait until user is ready before terminating program
  // to allow the user to see the program results
  system("PAUSE");
  return 0;
}
```

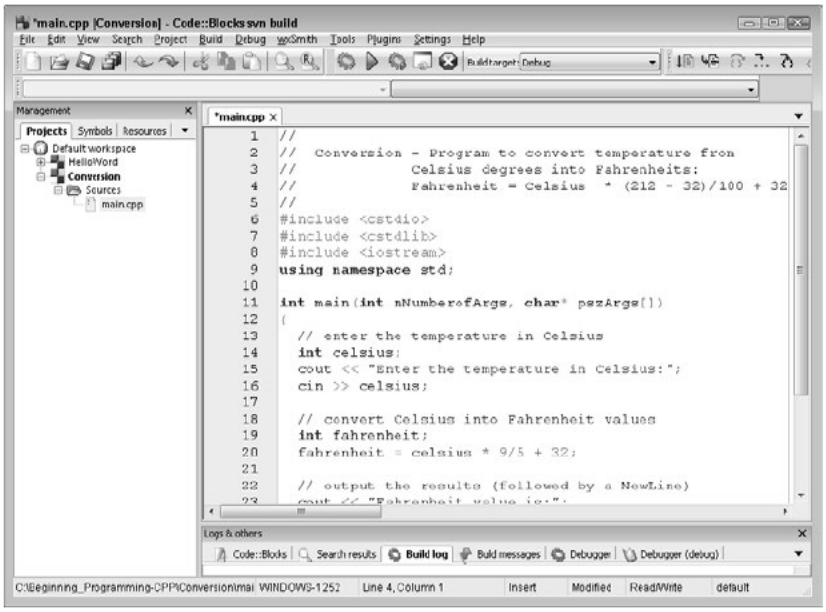


Figure 3-3: The edited `main.cpp` file of the Conversion program.

What do I mean by "exactly as you see here"? C++ is very picky about syntax. It frowns on missing semicolons or misspelled words. It doesn't care about extra spaces as long as they don't appear in the middle of a word. For example `int fahren heit;` is not the same as `int fahrenheit;` but `int fahrenheit;` is okay. C++ treats tabs, spaces, and newlines all the same, referring to them all as simply whitespace.

Maybe it was just me, but it took me a long time to get used to the fact that C++ differentiates between uppercase and lowercase. Thus, `int Fahrenheit;` is not the same thing as `int fahrenheit;`. One final hint: C++ ignores anything that appears after a `//`, so you don't have to worry about getting that stuff right.

6. **Save the file by selecting File➪Save all files.**

Building the Program

Now comes the most nerve-racking part of the entire software development process: building your program. It's during this step that C++ reviews your handiwork to see if it can make any sense out of what you've written.

Programmers are eternal optimists. Somewhere, deep in our hearts, we truly believe that every time we hit the Build button, everything is going to work, but it almost never does. Invariably, a missing semicolon or a misspelled word will disappoint C++ and bring a hail of error messages, like so much criticism from our elementary school teachers, crashing down around our ears.

Actually building the program takes just one step: You select Build➪Build or press Ctrl+F9 or click the little Build icon.

Finding What Could Go Wrong

No offense, but the Build step almost certainly did not come off without error. A Gold Star program is one that works the first time you build and execute it. You will almost never write a Gold Star program in your entire programming career.

Fortunately, the Code::Blocks editor is so well integrated with the compiler that it can automatically direct you very close to your errors. Most times, it can place the cursor in the exact row that contains the error. To prove the point, let me take you through a couple of example errors.

These are just two of the myriad ways to screw up in C++. I can't possibly show you all of them. Learning how to interpret what the compiler is trying to tell you with its error and warning messages is an important part of learning the language. It can come only from many months of practice and gaining experience with the language. Hopefully, these two examples will get you jump-started.

Misspelled commands

Misspelled commands are the easiest errors to identify and correct. To demonstrate the point, I added an extra `t` to line 14 in the preceding code so that it now reads

```
intt celsius;
```

Unlike `int`, the word `intt` has no meaning to C++. Building the resulting program generated the display shown in Figure 3-4.

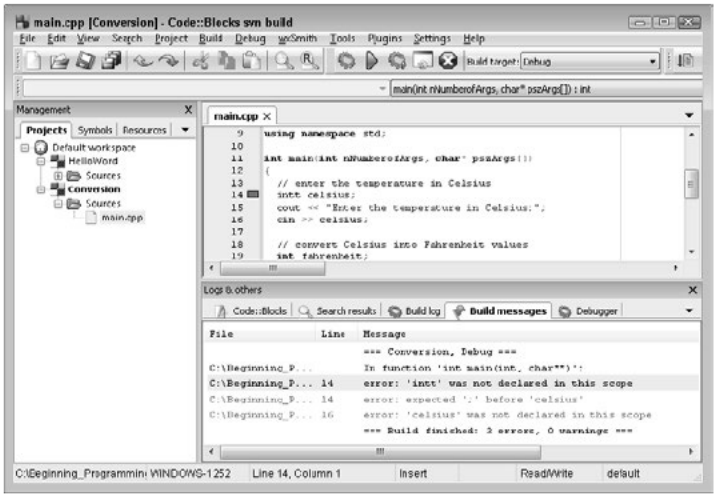


Figure 3-4: The error messages resulting from misspelling `int`.

Notice first the small, red block on Line 14 indicating that there is a problem somewhere on this line. You can read all about it down in the Build Messages tab in the lower-right window. Here you can see the following messages:

```
   In function 'int main(int, char**)':
14 error: 'intt' was not declared in this scope
14 error: expected ';' before 'celsius'
16 error: 'celsius' was not declared in this scope
```

The first line indicates the name of the function containing the error. I don't present functions until Chapter 12, but it's easy to believe that all of the code in this program is in a function called `main`. The next line is the key. This says essentially that C++ didn't understand what `intt` is on line 14 of the program. The error message is a bit cryptic, but suffice it to say you'll get this same error message almost every time you misspell something. The remaining error messages are just by-products of the original error.

One C++ error can generate a cascade of error messages. It is possible to identify and fix multiple errors in a single build attempt, but it takes experience to figure out which errors stem from which others. For now, focus on the first error message. Fix it and rebuild the program.

Why is C++ so picky?

You will quickly come to appreciate that C++ is about as picky as a judge at a spelling bee. Everything has to be just so, or the compiler won't accept it. Interestingly enough, it doesn't have to be that way: Some languages choose to try to make sense out of whatever you give it. The most extreme version of this was a language promulgated by IBM for its mainframes in the 1970s known as PL/1 (this stood for "Programming Language 1"). One version of this compiler would try to make sense out of whatever you threw at it. We nerds used to get immense fun during late nights at the computer center by torturing the compiler with a program consisting of nothing but the word "IF" or "WHILE." Through some tortured logic, PL/1 would construct an entire program out of this one command.

The other camp in programming languages, the camp to which C++ belongs, holds the opposite view: These languages compel the programmer to state exactly what she intends. Everything must be spelled out. Each declaration is checked against each and every usage to make sure that everything matches. No missing semicolon or incorrectly declared label goes unpunished.

It turns out that the tough love approach adopted by C++ is actually more efficient. The problem with the PL/1 "free love" approach is that it was almost always wrong in its understanding of what I intended. PL/1 ended up creating a program that compiled but did something other than what I intended when it executed. C++ generates a compiler error if something doesn't check out to force me to express my intentions clearly and unambiguously.

It turns out that it's a lot easier to find and fix the compile time errors generated by C++ than the so-called runtime errors created by a compiler that assumes it understands what I want but gets it wrong.

Missing semicolon

Another common error is to leave off a semicolon. The message that this error generates can be a little confusing. To demonstrate, I removed the semicolon from the declaration on line 14 so that it reads

```
int celsius
cout << "Enter the temperature in Celsius:";
```

The error reported by C++ for this offense points not to line 14 but to the following line 15:

```
15 error: expected initialization before 'cout'
16 error: 'celsius' was not declared in this scope
```

This is easier to understand when you consider that C++ considers newlines as just a different form of space. Without the semicolon, C++ runs the two lines together. There is no separate line 14 anymore. C++ can interpret the first part, but it doesn't understand the run-on sentence that starts with `cout`.

Missing semicolons often generate error messages that bear little resemblance to the actual error message, and they are almost always on the next line after the actual error. If you suspect a missing semicolon, start on the line with the reported error and scan backwards.

Using the Enclosed CD-ROM

If you just can't get the program entered correctly, you can always copy the program from the enclosed CD-ROM. (If you have questions regarding using the CD-ROM, see the Appendix; it details what you'll find on the CD-ROM, as well as troubleshooting tips, should you need them.)

You should really try to enter the program by hand first before you give up and resort to the CD-ROM. It's only through working through your mistakes that you develop a feel for how the language works.

You have several ways to use the enclosed CD-ROM. The most straightforward is to copy and paste the contents of the file on the CD into your own as follows:

1. **Insert the enclosed CD-ROM into your computer.**
2. **Select File⇨Open from within Code::Blocks. Navigate to the `X:\Beginning_Programming-CPP\Conversion` where *X* is the letter of your CD-ROM drive.**
3. **Select the file `main.cpp`.**

   Code::Blocks will open the file (in ReadOnly mode) in a new tab in the editor window.
4. **Select Edit⇨Select All or press Ctrl+A.**

   This will select the entire contents of the source file.
5. **Select Edit⇨Copy or press Ctrl+C.**

   This will copy the entire file to the clipboard.
6. **Select the main tab corresponding to your program.**
7. **Select Edit⇨Select All or press Ctrl+A again.**
8. **Select Edit⇨Paste or press Ctrl+V.**

   This will overwrite the entire contents of the `main.cpp` that you've been working on with the contents of the corresponding file on the CD-ROM.
9. **Close the tab containing the CD-ROM version of the file by clicking on the small X next to the filename.**

Running the Program

You can execute the program once you get a clean compile (that is, 0 errors and 0 warnings) by following these steps:

1. **Select Build⇨Run or press Ctrl+F10.**

   This will execute the program without the debugger. (Don't worry if you don't know what a *debugger* is; I teach you how to use it in Chapter 20.)

   The program opens an 80 column by 25 row window and prompts you to enter a temperature in degrees Celsius.

2. **Enter a known temperature like 100 degrees. Press Enter.**

   The program immediately responds with the equivalent temperature in Fahrenheit of 212:

```
Enter the temperature in Celsius:100
Fahrenheit value is:212
Press any key to continue . . .
```

3. **Press Enter twice to exit the program and return to the editor.**

How the Program Works

Even though this is your first program, I didn't want to leave this chapter without giving you some idea of how this program works.

The template

The first part of the program I call the "Beginning Programming Template." This will be the same magic incantation used for all programs in this book. It goes like this:

```
//
//  ProgramName - short explanation of what the
//                program does
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
```

```
{
  // program goes here

  // wait until user is ready before terminating program
  // to allow the user to see the program results
  system("PAUSE");
  return 0;
}
```

Comments

The first few lines in this template appear to be free-form text. Either this "code" was meant for human consumption or the computer is a lot smarter than anyone's ever given it credit for. These first four lines are known as *comments*. A comment is a line or portion of a line that is ignored by the C++ compiler. Comments enable the programmer to explain what she was doing or thinking while writing a particular segment of code.

A C++ comment begins with double forward slashes and ends with a newline. You can put any character you want in a comment, and comments can be as long as you like, though it is customary to limit them to 80 characters or so because that's what fits within a normal screen width.

Note: You may think it odd to have a command in C++, or any other programming language, that is specifically ignored by the compiler; yet, all programming languages have some form of comment. It is critical that the programmer be able to explain what was going through her mind when a piece of code was written. It may not be obvious to the next person who picks up the program and uses it or modifies it. In fact, it may not be obvious to the programmer herself after only a few days working on something else.

Include files

The next few lines are called *include statements* because they cause the contents of the named file to be included at that point in the program. Include files always start with the statement `#include` in column 1 followed by the name of the file to include. I'll explain further in Chapter 12. Just consider them magic for now.

main

Every program must have a `main()` somewhere in it. Program execution begins at the open brace immediately following `main()` and terminates at the return statement immediately prior to the closed brace. An explanation of the exact format of the declaration for `main()` will have to wait.

Notice that the standard template ends with the statement `system("PAUSE")` prior to the `return 0`. This command causes the program to wait for the user to enter a key before the program terminates.

The call to `system("PAUSE")` isn't necessary as long as you're running your programs from the Code::Blocks environment. Code::Blocks waits for the user to enter a key before closing the console application window anyway. However, not all environments are so understanding. Leave this off and very often C++ will close the application window before you have a chance to read the output from the program. I get lots of hate mail when that happens.

The Conversion program

The remainder of the `Conversion` program sans the template appears as follows:

```
// enter the temperature in Celsius
int celsius;
cout << "Enter the temperature in Celsius:";
cin >> celsius;

// convert Celsius into Fahrenheit values
int fahrenheit;
fahrenheit = celsius * 9/5 + 32;

// output the results (followed by a NewLine)
cout << "Fahrenheit value is:";
cout << fahrenheit << endl;
```

Skipping over the comment lines, which C++ ignores anyway, this program starts by declaring a variable called `celsius`. A variable is a place you can use to store a number or character.

The next line displays the prompt to the user to `"Enter the temperature in Celsius:"`. The object `cout` points to the console window by default.

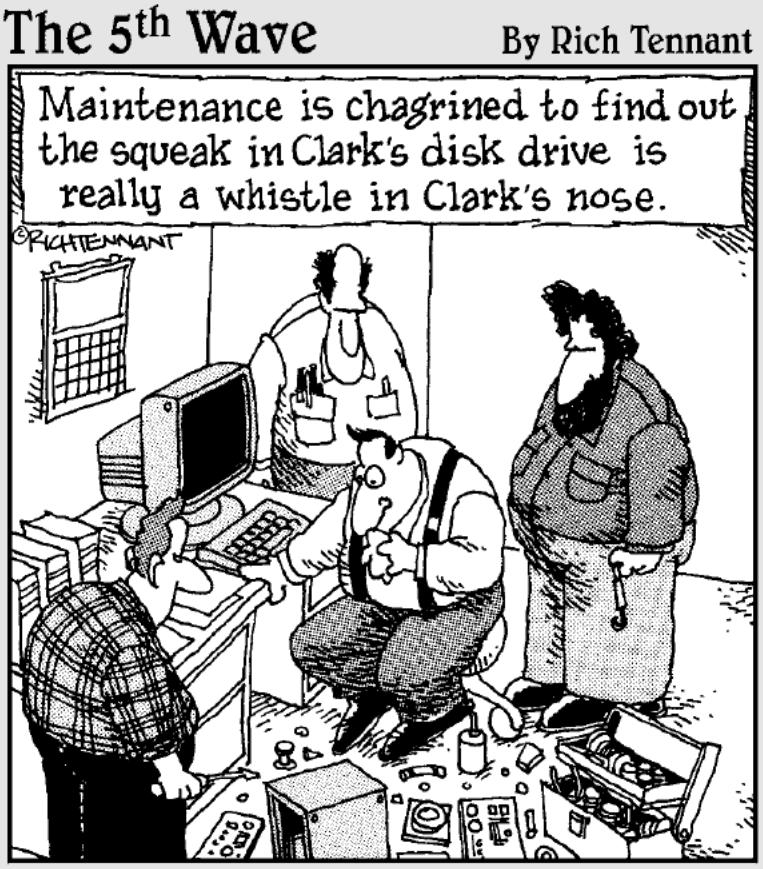
The next line reads whatever number the operator enters and stores it into the variable `celsius` declared earlier.

The next two lines declare a second variable `fahrenheit`, which it then sets equal to the value of the variable `celsius * 9 / 5 + 32`, which is the conversion formula from Celsius to Fahrenheit temperature.

The final two lines output the string `"Fahrenheit value is:"` and the value calculated and stored into the variable `fahrenheit` immediately above.

Part II

Writing a Program: Decisions, Decisions



In this part . . .

Now that you're familiar with how to write and build a program, you can start learning about C++ itself. This part introduces you to the basic elements of C++: the variable declaration and the expression. You'll even find out how to make a decision in your program ~~if~~ you can stand it. Finally, you'll see some beginning techniques for finding errors in your programs.

Chapter 4

Integer Expressions

In This Chapter

- Declaring variables
- Creating expressions
- Decomposing compound expressions
- Analyzing the assignment operator
- Incrementing variables with the unary operator

In this chapter, you will be studying integer declarations and expressions. Algebra class introduced you to the concepts of variables and expressions. The teacher would write something on the board like

```
x = 1
```

This defines a *variable* `x` and sets it equal to the value 1 until some subsequent statement changes it for some reason. The term `x` becomes a replacement for 1. The teacher would then write the following *expression*:

```
y = 2x
```

Because I know that `x` is 1, I now know that `y` is equal to 2. This was a real breakthrough in the seventh grade. All conventional computer languages follow this same pattern of creating and manipulating variables.

Declaring Variables

An integer *variable declaration* starts with the keyword `int` followed by the name of a variable and a semicolon, as in the following example:

```
int n1;       // declare a variable n1
```

All variables in C++ must be declared before they can be used. A variable declaration reserves a small amount of space in memory, just enough for a single integer, and assigns it a name. You can declare more than one variable in the same declaration, as in the following example, but it's not a good idea for reasons that will become clear as you work through subsequent chapters:

```
int n2, n3;   // declare two variables n2 and n3
```

A *keyword* is a word that has meaning to C++. You cannot name a variable the same as a keyword. Thus, you cannot create a variable with the name `int`. However, since keywords are case-sensitive, you could create a variable `Int` or `INT`. You will be introduced to further keywords throughout the chapters.

The fact that the keyword `int` is used instead of `integer` is just a reflection of the overall terseness of the C++ language. The creators of the language must have been poor typists and wanted to minimize the amount of typing they had to do.

Unlike in algebra class, the range of an integer in C++ is not unlimited. However, it is very large indeed. If you exceed the range of an int, you will get the wrong answer. I will discuss variable size and range in Chapter 14.

Variable names

You can name a variable anything you like with the following restrictions:

- The first letter of the variable must be a character in the sequence a through z, A through Z, or underscore ('_').
- Every letter after the first must be a character in the sequence a through z, A through Z, underscore ('_'), or the digits 0 through 9.
- A variable name can be of any length. All characters are significant.

The following are legal variable names:

```
int myVariable;
int MyVariable;
int myNumber2Variable;
int _myVariable;
int my_Variable;
```

The following are not legal variable names:

```
int myPercentage%;      // contains illegal character
int 2ndVariable;        // starts with a digit
int my Variable;        // contains a space
```

Variable names should be descriptive. Variable names like `x` are discouraged.

Assigning a value to a variable

Every variable has a value from the moment it's declared. However, until you assign it a value, a variable will just assume whatever garbage value happens to be in that memory location when it's allocated. That means that you don't know what the value is, and it's likely to change every time you run the program.

You can assign a variable a value using the equals sign as in the following example:

```
int n;          // declare a variable n
n = 1;          // set it to 1
```

This looks remarkably similar to the assignment statement in algebra class, but the effect is not quite the same. In C++, the assignment statement says "take the value on the right-hand side of the equals sign" (in this case 1) "and store it into the location on the left-hand side, overwriting whatever was there before" (in this case `n`).

You can see the difference in the following expression:

```
n = n + 1;      // increment the variable n
```

This statement would make absolutely no sense in algebra class. How could `n` be both equal to `n` and `n + 1` at the same time? However, this statement makes perfect sense in C++ if you follow the definition for assignment given above: "Take the value stored in the variable `n`" (1) "add 1 and store the result" (2) "into the variable `n`." This is shown graphically in Figure 4-1.

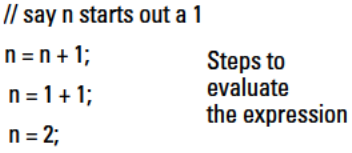


Figure 4-1: The effect of executing the expression `n = n + 1` when n starts out as 1.

Initializing a variable at declaration

You can initialize your variable at the time that it's declared by following it with an equals sign and a value:

```
int n = 1;      // declare and initialize variable
```

Forgetting to initialize a variable

Forgetting to initialize a variable before using it is a very common error in C++. So much so that the compiler actually goes to great pains to detect this case and warn you about it. Consider the following statements:

```
int n1, n2 = 0;
n2 = n1 + 1;
cout << "n1 = " << n1 << endl;
cout << "n2 = " << n2 << endl;
```

CodeBlocks generates the following warning when building the program containing this snippet:

```
warning: "n1" is used uninitialized in this function
```

Though it's a really bad idea, you are free to ignore warnings. Executing the program generates the output:

```
n1 = 54
n2 = 55
```

It's easy to see why `n2` is equal to 55 given that `n1` is 54, but why is `n1` equal to 54 in the first place? I could turn the question around and ask, "Why not?" This is an expression of the old adage, "Everyone has to be somewhere." The C++ equivalent is, "Every variable must have a value." If you don't initialize a variable to something, it'll get a random value from memory. In this case, the value 54 was left over from some previous usage.

This initializes only the one variable, so if you write the following compound declaration

```
int n1, n2 = 0;
```

you've initialized `n2` but not `n1`. This is one reason it's not a good idea to declare multiple variables in a single declaration. (See the sidebar "Forgetting to initialize a variable".)

Integer Constants

C++ understands any symbol that begins with a digit and contains only digits to be an *integer constant*. The following are legal constants:

```
123
1
256
```

A constant cannot contain any funny characters. The following is not legal:

```
123Z456
```

The following is legal but doesn't mean what you think:

```
123+456
```

This actually defines the sum of two constants `123` and `456`, or the value `479`.

Normally C++ assumes that constants are decimal (base 10). However, for historical reasons, a number that begins with a 0 is assumed to be octal (base 8). By the same token, a number that starts with `0x` or `0X` is assumed to be hexadecimal. Hexadecimal uses the letters `A` through `F` or `a` through `f` for the digits beyond 9. Thus, `0xFF`, `0377`, and `255` are all equivalent. Don't worry if you don't know what octal or hexadecimal are — we won't be using them in this book.

Don't start a constant with 0 unless you mean it to be in octal.

An integer constant can have certain symbols appended to the end to change its type. You will see the different types of integer constants in Chapter 14.

Expressions

Variables and constants are useful only if you can use them to perform calculations. The term *expression* is C++ jargon for a calculation. You've already seen the simplest expression:

```
int n;                  // declaration
n = 1;                  // expression
```

Expressions always involve variables, constants, and operators. An *operator* performs some arithmetic operation on its arguments. Most operators take two arguments — these are called *binary operators*. A few operators take a single argument — these are the *unary operators*.

All expressions return a value and a type. (Note that `int` is the type of all the expressions described in this chapter.)

Binary operators

A *binary operator* is an operator that takes two arguments. If you can say `var1 op var2`, then `op` must be a binary operator. The most common binary operators are the same simple operations that you learned in grade school. The common binary operators appear in Table 4-1. (This table also includes the unary operators that are described a little later in this chapter.)

Table 4-1 Mathematical Operators in Order of Precedence

Precedence	*Operator*	*Meaning*
1	- (unary)	Returns the negative of its argument
2	++ (unary)	Increment
2	-- (unary)	Decrement
3	* (binary)	Multiplication
3	/ (binary)	Division
3	% (binary)	Modulo
4	+ (binary)	Addition
4	- (binary)	Subtraction
5	=, *=,%=,+=,-= (special)	Assignment types

The simplest binary is the assignment operator noted by the equals sign. The assignment operator says "take the value on the right-hand side and store at the location on the left-hand side of the operator." (I describe the special assignment operators at the end of this chapter.)

Multiplication, division, addition, subtraction, and modulo are the operators used to perform arithmetic. They work just like the arithmetic operators you learned in grammar school with the following special considerations:

- **Multiplication must always be expressly stated and is never implied as it is in algebra.** Consider the following example:

```
int n = 2;                  // declare a variable
int m = 2n;                 // this generates an error
```

  The expression above does not assign m the value of 2 times n. Instead, C++ tries to interpret 2n as a variable name. Since variable names can't start with a digit, it generates an error during the build step.

  What the programmer meant was:

```
int n = 2;
int m = 2 * n;              // this is OK
```

- **Integer division throws away the remainder.** Thus, the following:

```
int n = 13 / 7;             // assigns the value 1 to n
```

  Fourteen divided by 7 is 2. Thirteen divided by seven is 1. (You will see decimal variable types that can handle fractions in Chapter 14.)

- **The modulo operator returns the remainder after division** (you might not remember modulo):

```
int n = 13 % 7;          // sets n to 6
```

Fourteen modulo seven is zero. Thirteen modulo seven is six.

Decomposing compound expressions

A single expression can include multiple operators:

```
int n = 5 + 100 + 32;
```

When all the operators are the same, C++ evaluates the expression from left to right:

```
5 + 100 + 32
105 + 32
137
```

When different operators are combined in a single expression, C++ uses a property called *precedence*. Precedence is the order that operators are evaluated in a compound expression. Consider the following example:

```
int n = 5 * 100 + 32;
```

What comes first, multiplication or addition? Or is this expression simply evaluated from left to right? Refer back to Table 4-1, which tells you that multiplication has a precedence of 3, which is higher than the precedence of addition which is 4 (smaller values have higher precedence). Thus, multiplication occurs first:

```
5 * 100 + 32
500 + 32
532
```

The order of the operations is overruled by the precedence of the operators. As you can see

```
int n = 32 + 5 * 100;
```

generates the same result:

```
32 + 5 * 100
32 + 500
532
```

But what if you really want 5 times the sum of 100 plus 32? You can override the precedence of the operators by wrapping expressions that you want performed first in parentheses as follows:

```
int n = 5 * (100 + 32);
```

Now the addition is performed before the multiplication:

```
5 * (100 + 32)
5 * 132
660
```

You can combine parentheses to make expressions as complicated as you like. C++ always starts with the deepest nested parentheses it can find and works its way out.

```
(3 + 2) * ((100 / 20) + (50 / 5))
(3 + 2) * (5 + 10)
5 * 15
75
```

You can always divide complicated expressions using intermediate variables. The following is safer:

```
int factor = 3 + 2;
int principal = (100 / 20) + (50 / 5);
int total = factor * principal;
```

Assigning a name to intermediate values also allows the programmer to explain the parts of a complex equation, making it easier for the next guy to understand.

Unary Operators

The *unary operators* are those operators that take a single argument. The unary mathematical operators are -, ++, and --.

The minus operator changes the sign of its argument. A positive number becomes negative, and a negative number becomes positive:

```
int n = 10;
int m = -n;              // m is now -10
```

The ++ and the -- operators increment and decrement their arguments by one.

Why a separate increment operator?

Why did the authors of C++ think that an increment operator was called for? After all, this operator does nothing more than add 1, which can be done with an assignment expression. The authors of C++ (and its predecessor C) were obsessed with efficiency. They wanted to generate the fastest machine code they possibly could. They knew that most processors have an increment and decrement instruction, and they wanted the C++ compiler to use that instruction if at all possible. They reasoned that `n++` would get converted into an increment instruction while `n = n + 1;` might not. This type of thing makes very little difference today, but the increment and decrement operators are here to stay. As you will see in Chapters 9 and 10, they get a lot more use than you might think.

The increment and decrement operators are unique in that they come in two versions: a *prefix* and a *postfix* version.

The prefix version of increment is written `++n`, while the postfix is written `n++`.

Both the prefix and postfix increment operators increment their argument by one. The difference is in the value returned. The prefix version returns the value after the increment operation, while the postfix returns the value before the increment. (The same is true of the decrement operator.) This is demonstrated in the following IncrementOperator program:

```
// IncrementOperator - demonstrate the increment operator

#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // demonstrate the increment operator
    int n;

    // first the prefix
    n = 1;
    cout << "The value of n is   " <<   n << endl;
    cout << "The value of ++n is " << ++n << endl;
    cout << "The value of n afterwards is " << n << endl;
    cout << endl;

    // now the postfix
    n = 1;
    cout << "The value of n is   " <<   n << endl;
    cout << "The value of n++ is " << n++ << endl;
```

```
    cout << "The value of n afterwards is " << n << endl;
    cout << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The output from this program appears as follows:

```
The value of n is   1
The value of ++n is 2
The value of n afterwards is 2

The value of n is   1
The value of n++ is 1
The value of n afterwards is 2

Press any key to continue . . .
```

This example demonstrates both the prefix and postfix increment. In both cases, the variable `n` is initialized to 1. Notice that the value of `n` after executing both `++n` and `n++` is 2. However, the value of `++n` was 2 (the value after the increment), while the value of `n++` was 1 (the value before the increment).

The Special Assignment Operators

The assignment operator is absolutely critical to any computer language. How else can I store a computed value? However, C++ provides a complete set of extra versions of the assignment operator that seems less critical.

The authors of C++ must have noticed that expressions of the following form were very common:

```
x = x # value;
```

Here # stands for some binary operator. In their perhaps overzealous pursuit of terseness, the authors created a separate assignment for each of the binary operators of the form:

```
x #= value;  // where # is any one of the binary operators
```

Thus, for example

```
n = n + 2;
```

can be written as

```
n += 2;
```

Note: You don't see this all that often, and I present it here primarily for completeness.

Chapter 5

Character Expressions

In This Chapter

- Defining character variables and constants
- Encoding characters
- Declaring a string
- Outputting characters to the console

Chapter 4 introduces the concept of the integer variable. This chapter introduces the integer's smaller sibling, the character or `char` (pronounced variously as *care*, *chair*, or as in the first syllable of *char*coal) to us insiders. I have used characters in programs appearing in earlier chapters — now it's time to introduce them formally.

Defining Character Variables

Character variables are declared just like integers except with the keyword `char` in place of `int`:

```
char inputCharacter;
```

Character constants are defined as a single character enclosed in single quotes, as in the following:

```
char letterA = 'A';
```

This may seem like a silly question, but what exactly is "A"? To answer that, I need to explain what it means to encode characters.

Encoding characters

As I mentioned in Chapter 1, everything in the computer is represented by a pattern of ones and zeros that can be interpreted as numbers. Thus, the bit pattern `0000 0001` is the number 1 when interpreted as an integer. However, this same bit pattern means something completely different when interpreted as an instruction by the processor. So it should come as no surprise that the computer encodes the characters of the alphabet by assigning each a number.

Consider the character 'A'. You could assign it any value you want as long as we all agree. For example, you could assign a value of 1 to 'A', if you wanted to. Logically, you might then assign the value 2 to 'B', 3 to 'C', and so on. In this scheme, 'Z' would get the value 26. You might then start over by assigning the value 27 to 'a', 28 to 'b', right down to 52 for 'z'. That still leaves the digits '0' through '9' plus all the special symbols like space, period, comma, slash, semicolon, and the funny characters you see when you press the number keys while holding Shift down. Add to that the unprintable characters like tab and newline. When all is said and done, you could encode the entire English keyboard using numbers between 1 and 127.

I say "you *could*" assign a value for 'A', 'B', and the remaining characters; however, that wouldn't be a very good idea because it has already been done. Sometime around 1963, there was a general agreement on how characters should be encoded in English. The ASCII (American Standard Coding for Information Interchange) character encoding shown in Table 5-1 was adopted pretty much universally except for one company. IBM published its own standard in 1963 as well. The two encoding standards duked it out for about ten years, but by the early 1970s when C and C++ were being created, ASCII had just about won the battle. The `char` type was created with ASCII character encoding in mind.

Table 5-1 The ASCII Character Set

Value	*Char*	*Value*	*Char*
0	NULL	64	@
1	Start of Heading	65	A
2	Start of Text	66	B
3	End of Text	67	C
4	End of Transmission	68	D
5	Enquiry	69	E

Value	*Char*	*Value*	*Char*
6	Acknowledge	70	F
7	Bell	71	G
8	Backspace	72	H
9	Tab	73	I
10	Newline	74	J
11	Vertical Tab	75	K
12	New Page; Form Feed	76	L
13	Carriage Return	77	M
14	Shift Out	78	N
15	Shift In	79	O
16	Data Link Escape	80	P
17	Device Control 1	81	Q
18	Device Control 2	82	R
19	Device Control 3	83	S
20	Device Control 4	84	T
21	Negative Acknowledge	85	U
22	Synchronous Idle	86	V
23	End of Transmission	87	W
24	Cancel	88	X
25	End of Medium	89	Y
26	Substitute	90	Z
27	Escape	91	[
28	File Separator	92	\
29	Group Separator	93	]
30	Record Separator	94	^
31	Unit Separator	95	_
32	Space	96	`
33	!	97	a
34	“	98	b
35	#	99	c
36	$	100	d
37	%	101	e

(continued)

Table 5-1 ***(continued)***

Value	*Char*	*Value*	*Char*
38	&	102	f
39	‘	103	g
40	(	104	h
41	)	105	i
42	*	106	j
43	+	107	k
44	,	108	l
45	=	109	m
46	.	110	n
47	/	111	o
48	0	112	p
49	1	113	q
50	2	114	r
51	3	115	s
52	4	116	t
53	5	117	u
54	6	118	v
55	7	119	w
56	8	120	x
57	9	121	y
58	:	122	z
59	;	123	{
60	<	124	|
61	=	125	}
62	>	126	~
63	?	127	DEL

The first thing that you’ll notice is that the first 32 characters are the “unprintable” characters. That doesn’t mean that these characters are so naughty that the censor won’t allow them to be printed — it means that they don’t display as a symbol when printed on the printer (or on the console for that matter). Many of these characters are no longer used or only used

in obscure ways. For example, character 25 "End of Medium" was probably printed as the last character before the end of a reel of magnetic tape. That was a big deal in 1963, but today it has limited use. My favorite is character 7, the Bell — this used to ring the bell on the old teletype machines. (The Code::Blocks C++ generates a beep when you display the bell character.)

The characters starting with 32 are all printable with the exception of the last one, 127, which is the Delete character.

Example of character encoding

The following simple program allows you to play with the ASCII character set:

```
// CharacterEncoding - allow the user to enter a
//                     numeric value then print that value
//                     out as a character

#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // Prompt the user for a value
    int nValue;
    cout << "Enter decimal value of char to print:";
    cin >> nValue;

    // Now print that value back out as a character
    char cValue = (char)nValue;
    cout << "The char you entered was [" << cValue
         << "]" << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This program begins by prompting the user to `"Enter decimal value of a char to print"`. The program then reads the value entered by the user into the `int` variable `nValue`.

The program then assigns this value to a `char` variable `cValue`.

The `(char)` appearing in front of `nValue` is called a *cast*. In this case, it casts the value of `nValue` from an `int` to a `char`. I could have performed the assignment without the cast as in

```
cValue = nValue;
```

However, the type of the variables wouldn't match: The value on the right of the assignment is an `int`, while the value on the left is a `char`. C++ will perform the assignment anyway, but it will generally complain about such conversions by generating a warning during the build step. The cast converts the value in `nValue` to a char before performing the assignment:

```
cValue = (char)nValue;  // cast nValue to a char before
                        // assigning the value to cValue
```

The final line outputs the character `cValue` within a set of square brackets.

The following shows a few sample runs of the program. In the first run, I entered the value 65, which Table 5-1 shows as the character 'A':

```
Enter decimal value of char to print:65
The char you entered was [A]
Press any key to continue . . .
```

The second time I entered the value 97, which corresponds to the character 'a':

```
Enter decimal value of char to print:97
The char you entered was [a]
Press any key to continue . . .
```

On subsequent runs, I tried special characters:

```
Enter decimal value of char to print:36
The char you entered was [$]
Press any key to continue . . .
```

The value 7 didn't print anything, but did cause my PC to issue a loud beep that scared the heck out of me.

The value 10 generated the following odd output:

```
Enter decimal value of char to print:10
The char you entered was [
]
Press any key to continue . . .
```

Referring to Table 5-1, you can see that 10 is the newline character. This character doesn't actually print anything but causes subsequent output to start

at the beginning of the next line, which is exactly what happened in this case: The closed brace appears by itself at the beginning of the next line when following a newline character.

The `endl` that appears at the end of many of the output commands that you've seen so far generates a newline. It also does a few other things, which you'll see in Chapter 31.

Encoding Strings of Characters

Theoretically, you could print anything you want using individual characters. However, that could get really tedious as the following code snippet demonstrates:

```
cout << 'E' << 'n' << 't' << 'e' << 'r' << ' '
     << 'd' << 'e' << 'c' << 'i' << 'm' << 'a'
     << 'l' << ' ' << 'v' << 'a' << 'l' << 'u'
     << 'e' << ' ' << 'o' << 'f' << ' ' << 'c'
     << 'h' << 'a' << 'r' << ' ' << 't' << 'o'
     << ' ' << 'p' << 'r' << 'i' << 'n' << 't'
     << ':';
```

C++ allows you to encode a sequence of characters by enclosing the string in double quotes:

```
cout << "Enter decimal value of char to print:";
```

I'll have a lot more to say about character strings in Chapter 16.

Special Character Constants

You can code a normal, printable character by placing it in single quotes:

```
char cSpace = ' ';
```

You can code any character you want, whether printable or not, by placing its octal value after a backslash:

```
char cSpace = '\040';
```

A constant appearing with a leading zero is assumed to be octal, also known as base 8.

You can code characters in base 16, hexadecimal, by preceding the number with a backslash followed by a small x as in the following example:

```
char cSpace = '\x20';
```

The decimal value 32 is equal to 40 in base 8 and 20 in base 16. Don't worry if you don't feel comfortable with octal or hexadecimal. C++ provides shortcuts for the most common characters.

C++ provides a name for some of the unprintable characters that are particularly useful. Some of the more common ones are shown in Table 5-2.

Table 5-2 Some of the Special C++ Characters

Char	*Special Symbol*	*Char*	*Special Symbol*
‘	\’	Newline	\n
“	\”	Carriage Return	\r
\	\\	Tab	\t
NULL	\0	Bell	\a

The most common is the newline character, which is nicknamed ‘\n’. In addition, you must use the backslash if you want to print the single quote character:

```
char cQuote = '\'';
```

Since C++ normally interprets a single quote mark as enclosing a character, you have to precede a single quote mark with a backslash character to tell it, “Hey, this single quote is not enclosing a character, this is the character.”

In addition, the character ‘\\’ is a single backslash.

This leads to one of the more unfortunate coincidences in C++. In Windows, the backslash is used in filenames as in the following:

```
C:\\Base Directory\Subdirectory\File Name
```

This is encoded in C++ with each backslash replaced by a pair of backslashes as follows:

```
"C:\\\\Base Directory\\Subdirectory\\File Name"
```

Wide load ahead

By the early 1970s when C and C++ were invented, the 128-character ASCII character set had pretty much beat out all rivals. So it was logical that the `char` type was defined to accommodate the ASCII character set. This character set was fine for English but became overly restrictive when programmers tried to write applications for other European languages.

Fortunately, C and C++ had provided enough room in the `char` for 256 different characters. Standards committees got busy and used the characters between 128 and 255 for characters that occur in European languages but not English, such as umlauts and accented characters. You can see the results of their handy work using the example `CharacterEncoding` program from this chapter: Enter 142 and the program prints out an Ä.

No matter what you do, the `char` variable is just not large enough to handle all of the many different alphabets, such as Cyrillic, Hebrew, Arabic, and Korean — not to mention the many thousands of Chinese kanji symbols. Something had to give.

C++ responded first by introducing the "wide character" of type `wchar_t`. This was intended to implement whatever wide character set that is native to the host operating system. On Windows, that would be the variant of Unicode known as UTF-2 or UTF-16. (Here the 2 stands for two bytes, the size of each wide character, whereas the 16 stands for 16 bits.) However, Macintosh's OS X uses a different variant of Unicode known as UTF-8. Unicode can display not only every alphabet on the planet but also the kanjis used in Chinese and Japanese. The 2009 update to the C++ standard added two further types, `char16_t` and `char32_t`, which implement specifically UTF-16 and UTF-32.

For almost every feature that I describe in this book for handling character variables, there is an equivalent feature for the wide character types; programming Unicode, however, is beyond the scope of a beginning text.

Chapter 6

if I Could Make My Own Decisions

In This Chapter

- Defining character variables and constants
- Encoding characters
- Declaring a string
- Outputting characters to the console

Making decisions is a part of the everyday world. Should I get a drink now or wait for the commercial? Should I take this highway exit to go to the bathroom or else wait for the next? Should I take another step or stop and smell the roses? If I am hungry or I need gas, then I should stop at the convenience store. If it is a weekend and I feel like it, then I can sleep in. See what I mean?

An assistant, even a stupid one, has to be able to make at least rudimentary decisions. Consider the Tire Changing Language in Chapter 1. Even there, the program had to be able to test for the presence of a lug nut to avoid waving a wrench around uselessly in space over an empty bolt, thereby wasting everyone's time.

All computer languages provide some type of decision-making capability. In C++, this is handled primarily by the `if` statement.

The if Statement

The format of the `if` statement is straightforward:

```
if (m > n)    // if m is greater than n...
{
              // ...then do this stuff
}
```

When encountering `if`, C++ first executes the logical expression contained within the parentheses. In this case, the program evaluates the conditional expression "is `m` greater than `n`." If the expression is `true`, that is, if `m` truly is greater than `n`, then control passes to the first statement after the `{` and continues from there. If the logical expression is not true, control passes to the first statement after the `}`.

Comparison operators

Table 6-1 shows the different operators that can be used to compare values in logical expressions.

Binary operators have the format `expr1 operator expr2`.

Table 6-1 The Comparison Operators

Operator	*Meaning*
==	equality; true if the left-hand argument has the same value as the expression on the right
!=	inequality; opposite of equality
>	greater than; true if the left-hand argument is greater than the right
<	less than; true if the left-hand argument is less than the right
>=	greater than or equal to; true if the left argument is greater than or equal to the right
<=	less than or equal to; true if the left argument is less than or equal to the right

Don't confuse the equality operator (==) with the assignment operator (=). This is a common mistake for beginners.

The following BranchDemo program shows how the operators shown in Table 6-1 are used:


```
// BranchDemo - demonstrate the if statement

#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
```

```
{
    // enter operand1 and operand2
    int  nOperand1;
    int  nOperand2;
    cout << "Enter argument 1:";
    cin  >> nOperand1;
    cout << "Enter argument 2:";
    cin  >> nOperand2;

    // now print the results
    if (nOperand1 > nOperand2)
    {
        cout << "Argument 1 is greater than argument 2"
             << endl;
    }
    if (nOperand1 < nOperand2)
    {
        cout << "Argument 1 is less than argument 2"
             << endl;
    }
    if (nOperand1 == nOperand2)
    {
        cout << "Argument 1 is equal to argument 2"
             << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

Program execution begins with `main()` as always. The program first declares two `int` variables cleverly named `nOperand1` and `nOperand2`. It then prompts the user to `"Enter argument 1"`, which it reads into `nOperand1`. The process is repeated for `nOperand2`.

The program then executes a sequence of three comparisons. It first checks whether `nOperand1` is less than `nOperand2`. If so, the program outputs the notification `"Argument 1 is less than argument 2"`. The second `if` statement displays a message if the two operands are equal in value. The final comparison is true if `nOperand1` is greater than `nOperand2`.

The following shows a sample run of the BranchDemo program:

```
Enter argument 1:5
Enter argument 2:10
Argument 1 is less than argument 2
Press any key to continue . . .
```

Figure 6-1 shows the flow of control graphically for this particular run.

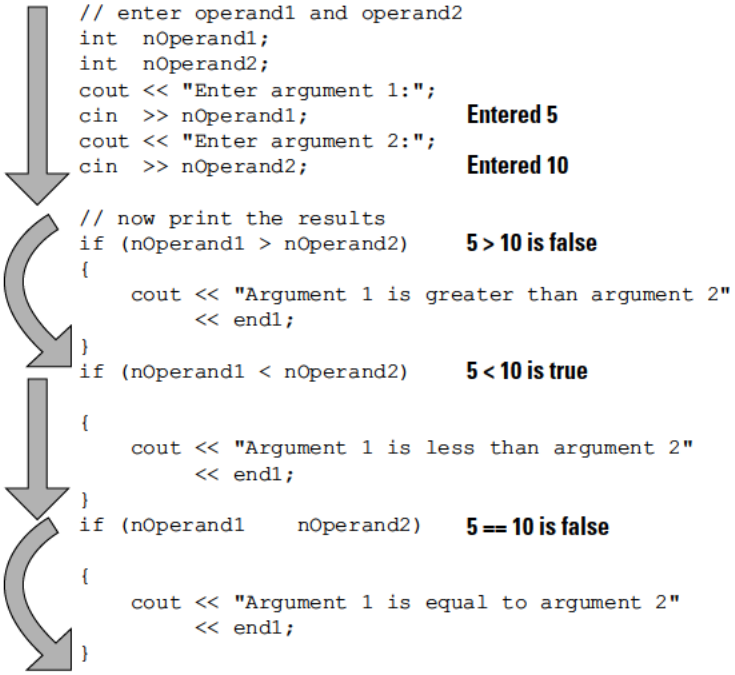


Figure 6-1: The path taken by the BranchDemo program when the user enters 5 for the first argument and 10 for the second.

The way the BranchDemo program is written, all three comparisons are performed every time. This is slightly wasteful since the three conditions are mutually exclusive. For example, `nOperand1 > nOperand2` can't possibly be true if `nOperand1 < nOperand2` has already been found to be true. Later in this chapter, I show you how to avoid this waste.

Say "No" to "No braces"

Actually the braces are optional. Without braces, only the first expression after the `if` statement is conditional. However, it is much too easy to make a mistake this way, as demonstrated in the following snippet:

```
// Can't have a negative age. If age is less than zero...
if (nAge < 0)
    cout << "Age can't be negative; using 0" << endl;
    nAge = 0;

// program continues
```

You may think that if `nAge` is less than 0, this program snippet outputs a message and resets `nAge` to zero. In fact, the program sets `nAge` to zero no matter what its original value. The preceding snippet is equivalent to the following:

```
// Can't have a negative age. If age is less than zero...
if (nAge < 0)
{
    cout << "Age can't be negative; using 0" << endl;
}
    nAge = 0;

// program continues
```

It's clear from the comments and the indent that the programmer really meant the following:

```
// Can't have a negative age. If age is less than zero...
if (nAge < 0)
{
    cout << "Age can't be negative; using 0" << endl;
    nAge = 0;
}

// program continues
```

The C++ compiler can't catch this type of mistake. It's safer just to always supply the braces.

C++ treats all white space the same. It ignores the alignment of expressions on the page.

Always use braces to enclose the statements after an `if` statement, even if there is only one. You'll generate a lot fewer errors that way.

What else Is There?

C++ allows the program to specify a clause after the keyword `else` that is executed if the conditional expression is false, as in the following example:

```
if (m > n)    // if m is greater than n...
{
              // ...then do this stuff;...
}
else          // ...otherwise,...
{
              // ...do this stuff
}
```

The `else` clause must appear immediately after the close brace of the `if` clause. In use, the `else` appears as shown in the following snippet:

```
if (nAge < 0)
{
    cout << "Age can't be negative; using 0." << endl;
    nAge = 0;
}
else
{
    cout << "Age of " << nAge << " entered" << endl;
}
```

In this case, if `nAge` is less than zero, the program outputs the message `"Age can't be negative; using 0."` and then sets `nAge` to 0. This corresponds to the flow of control shown in the first image in Figure 6-2. If `nAge` is not less than zero, the program outputs the message `"Age of x entered"`, where *x* is the value of `nAge`. This is shown in the second image in Figure 6-2.

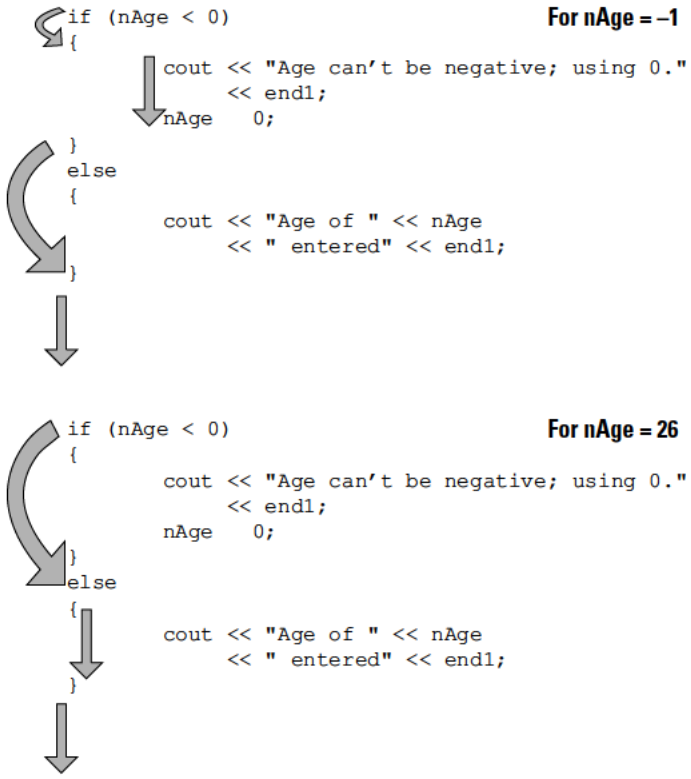


Figure 6-2: Flow of control through an `if` and `else` for two different values of `nAge`.

Logical expressions: Do they have any value?

At the beginning of this chapter, I called the comparison symbols < and > *operators,* and I described statements containing these operators as *expressions.* But expressions have a value and a type. What is the value and type of an expression like `m > n`? In C++, the type of this expression is `bool` (named in honor of George Boole, the inventor of Logic Calculus). Expressions of type `bool` can have only one of two values: `true` or else `false`. Thus, you can write the following:

```
bool bComparison = m > n;
```

For historical reasons, there is a conversion between the numerical types like `int` and `char` and `bool`: A value of 0 is considered the same as `false`. Any non-zero value is considered the same as `true`.

Thus, the `if` statement

```
if (cCharacter)
{
    // execute this code if cCharacter is not NULL
}
```

is the same as

```
if (cCharacter != '\0')
{
    // execute this code if cCharacter is not NULL
}
```

Assigning a true/false value to a character may seem a bit obtuse, but you'll see in Chapter 16 that it has a very useful application.

Nesting if Statements

The braces of an `if` or an `else` clause can contain another `if` statement. These are known as *nested* `if` statements. The following NestedIf program shows an example of a nested `if` statement in use.

```
// NestedIf - demonstrate a nested if statement
//
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
```

```
    // enter your birth year
    int  nYear;
    cout << "Enter your birth year: ";
    cin  >> nYear;

    // Make determination of century
    if (nYear > 2000)
    {
        cout << "You were born in the 21st century"
             << endl;
    }
    else
    {
        cout << "You were born in ";
        if (nYear < 1950)
        {
            cout << "the first half";
        }
        else
        {
            cout << "the second half";
        }
        cout << " of the 20th century"
             << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This program starts by asking the user for his birth year. If the birth year is greater than 2000, then the program outputs the string `"You were born in the 21st century"`.

The year 2000 belongs to the 20th century, not the 21st.

If the birth year is not greater than 2000, then the program enters the `else` clause of the outer `if` statement. This clause starts by outputting the string `"You were born in"` before comparing the birth year to 1950. If the birth year is less than 1950, then the program adds the first `"the first half"`. If the birth year is not less than 1950, then the `else` clause of the inner `if` statement is executed, which tacks on the phrase `"the second half"`. Finally, the program adds the concluding phrase `"of the 20th century"` to whatever has been output so far.

In practice, the output of the program appears as follows for three possible values for birth year. First, 2002 produces the following:

```
Enter your birth year: 2002
You were born in the 21st century
Press any key to continue . . .
```

My own birth year of 1956 generates the following:

```
Enter your birth year: 1956
You were born in the second half of the 20th century
Press any key to continue . . .
```

Finally, my father's birth year of 1932 generates the third possibility:

```
Enter your birth year: 1932
You were born in the first half of the 20th century
Press any key to continue . . .
```

I could use a nested `if` to avoid the unnecessary comparisons in the NestedBranchDemo program:

```
if (nOperand1 > nOperand2)
{
    cout << "Argument 1 is greater than argument 2"
         << endl;
}
else
{
    if (nOperand1 < nOperand2)
    {
        cout << "Argument 1 is less than argument 2"
             << endl;
    }
    else
    {
        cout << "Argument 1 is equal to argument 2"
             << endl;
    }
}
```

This version performs the first comparison just as before. If `nOperand1` is greater than `nOperand2`, this snippet outputs the string `"Argument 1 is greater than argument 2"`. From here, however, control jumps to the final closed brace, thereby skipping the remaining comparisons.

If nOperand1 is not greater than nOperand2, then the snippet performs a second test to differentiate the case that nOperand1 is less than nOperand2 from the case that they are equal in value.

Figure 6-3 shows graphically the flow of control for the NestedBranchDemo program for the same input of 5 and 10 described earlier in the chapter.

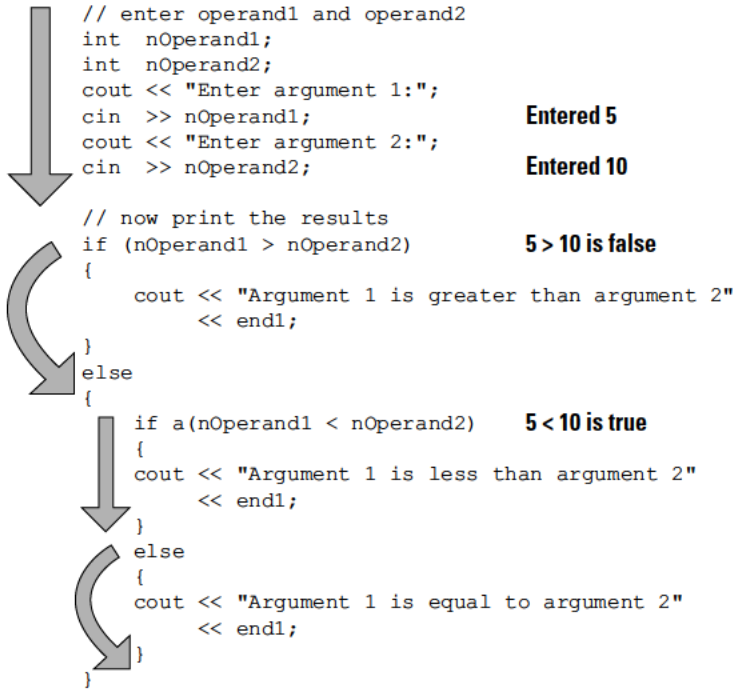


Figure 6-3: The path taken by the NestedBranchDemo program when the user enters 5 and 10 as before.

Performing the test for equality is unnecessary: If nOperand1 is neither greater than nor less than nOperand2, then it must be equal.

Compound Conditional Expressions

The three logical operators that can be used to create what are known as *compound conditional expressions* are shown in Table 6-2.

Table 6-2 The Logical Operators

Operator	*Meaning*
&&	AND; true if the left- and right-hand arguments are true; otherwise, false
||	OR; true if either the left- or right-hand arguments is true; otherwise, false
!	NOT; true if the argument on the right is false; otherwise, false

The programmer is asking two or more questions in a conditional compound expression, as in the following code snippet:

```
// make sure that nArgument is between 0 and 5
if (0 < nArgument && nArgument < 5)
```

Figure 6-4 shows how three different values of `nArgument` are evaluated by this expression.

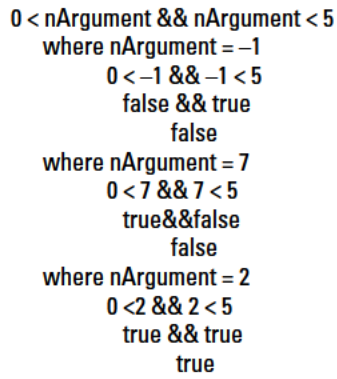


Figure 6-4: The evaluation of the compound expression `0 < n && n < 5` for three different values of `n`.

By the way, the snippet

```
if (m < nArgument && nArgument < n)
```

is the normal way of coding the expression `"if nArgument is between m and n, exclusive"`. This type of test does not include the end points — that is, this test will fail if `nArgument` is equal to `m` or `n`. Use the <= comparison operator if you want to include the end points.

Short circuit evaluation

Look carefully at a compound expression involving a logical AND like

```
if (expr1 && expr2)
```

If `expr1` is `false`, then the overall result of the compound expression is `false`, irrespective of the value of `expr2`. In fact, C++ doesn't even evaluate `expr2` if `expr1` is `false` — `false && anything` is `false`. This is known as *short circuit evaluation* because it short circuits around executing unnecessary code in order to save time.

The situation is exactly the opposite for the logical OR:

```
if (expr1 || expr2)
```

If `expr1` is `true`, then the overall expression is `true`, irrespective of the value of `expr2`.

Short circuit evaluation is a good thing since the resulting programs execute more quickly; however, it can lead to unexpected results in a few cases. Consider the following admittedly contrived case:

```
if (m <= nArgument && nArgument++ <= n)
```

The intent is to test whether `nArgument` falls into the range [m, n] and to increment `nArgument` as part of the test. However, short circuit evaluation means that the second test doesn't get executed if `m <= nArgument` is not `true`. If the second test is never evaluated, then `nArgument` doesn't get incremented.

Remember: If you didn't follow that, just remember the following: Don't put an expression that has a side effect like incrementing a variable in a conditional.

Chapter 7

Switching Paths

In This Chapter

- Using the `switch` keyword to choose between multiple paths
- Taking a `default` path
- Falling through from one case to another

Often programs have to decide between a very limited number of options: Either `m` is greater than `n` or it's not; either the lug nut is present or it's not. Sometimes, however, a program has to decide between a large number of possible legal inputs. This could be handled by a series of `if` statements, each of which tests for one of the legal inputs. However, C++ provides a more convenient control mechanism for selecting among a number of options known as the `switch` statement.

Controlling Flow with the switch Statement

The `switch` statement has the following format:

```
switch(expression)
{
  case const1:
    // go here if expression == const1
    break;

  case const2:
    // go here if expression == const2
    break;

  case const3:          // repeat as often as you like
    // go here if expression == const3
    break;

  default:
    // go here if none of the other cases match
}
```

Upon encountering the `switch` statement, C++ evaluates `expression`. It then passes control to the case with the same value as `expression`. Control continues from there to the `break` statement. The `break` transfers control to the `}` at the end of the `switch` statement. If none of the cases match, control passes to the default case.

The default case is optional. If the expression doesn't match any case and no default case is provided, control passes immediately to the `}`.

Consider the following example code snippet:

```
int nMonth;
cout << "Enter the number of the month: ";
cin  >> nMonth;

switch (nMonth)
{
  case  1:
    cout << "It's January"  << endl;
    break;
  case  2:
    cout << "It's February" << endl;;
    break;
  case  3:
    cout << "It's March"    << endl;;
    break;
  case  4:
    cout << "It's April"    << endl;;
    break;
  case  5:
    cout << "It's May"      << endl;;
    break;
  case  6:
    cout << "It's June"     << endl;;
    break;
  case  7:
    cout << "It's July"     << endl;;
    break;
  case  8:
    cout << "It's August"   << endl;;
    break;
  case  9:
    cout << "It's September"<< endl;;
    break;
  case 10:
    cout << "It's October"  << endl;;
    break;
  case 11:
    cout << "It's November" << endl;;
    break;
```

```
  case 12:
    cout << "It's December" << endl;;
    break;
  default:
    cout << "That's not a valid month" << endl;;

}
```

I got the following output from the program when inputting a value of 3:

```
Enter the number of the month: 3
It's March
Press any key to continue . . .
```

Figure 7-1 shows how control flowed through the `switch` statement to generate the earlier result of "March."

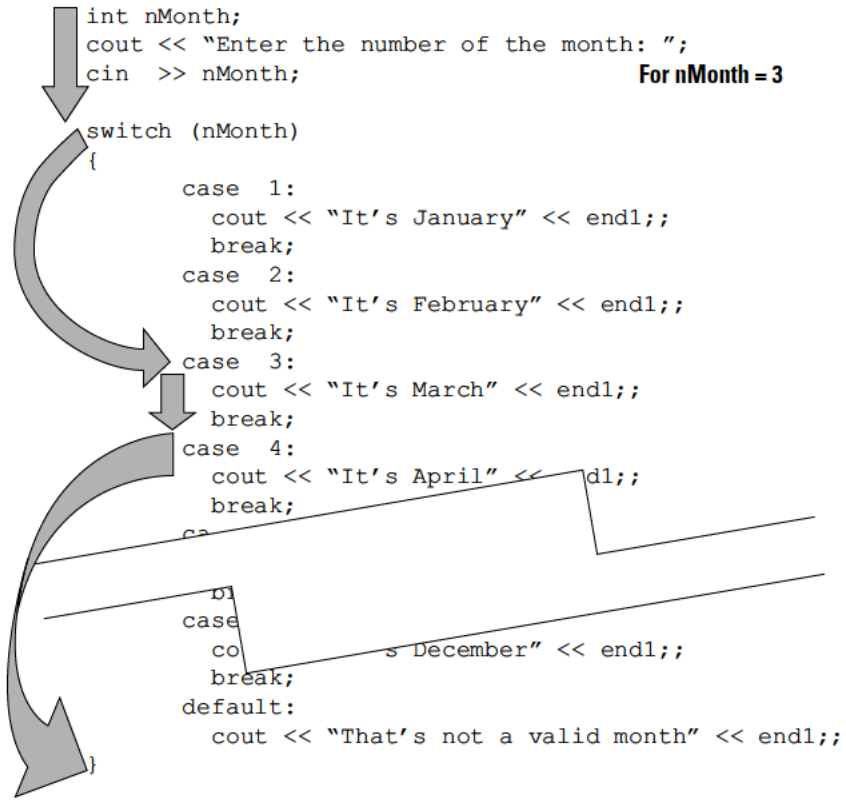


Figure 7-1: Flow through a `switch` statement listing the months of the year where the operator enters month 3.

A `switch` statement is not like a series of `if` statements. For example, only constants are allowed after the `case` keyword (or expressions that can be completely evaluated at build time). You cannot supply an expression after a case. Thus, the following is not legal:

```
// cases cannot be expressions; in general, the
// following is not legal
switch(n)
{
  case m:
    cout << "n is equal to m" << endl;
    break;
  case 2 * m:
    cout << "n is equal to 2m" << endl;
    break;
  case 3 * m:
    cout << "n is equal to 3m" << endl;
}
```

Each of the cases must have a value at build time. The value of `m` is not known until the program executes.

Control Fell Through: Did I break It?

Just as the default case is optional, the `break` at the end of each case is also optional. Without the `break` statement, however, control simply continues on from one case to the next. Programmers say that control *falls through*. This is most useful when two or more cases are handled in the same way.

For example, C++ may differentiate between upper- and lowercase, but most humans do not. The following code snippet prompts the user to enter a C to create a checking account and an S to create a savings account. However, by providing extra case statements, the snippet handles lowercase c and s the same way:

```
cout << "Enter C to create checking account, "
     << "S to create a saving account, "
     << "and X to exit: ";
cin  >> cAccountType;
switch(cAccountType)
{
  case 'S':         // upper case S
  case 's':         // lower case s
     // creating savings account
     break;

  case 'C':         // upper case C
  case 'c':         // lower case c
```

```
    // create checking account
    break;

  case 'X':         // upper case X
  case 'x':         // lower case x
    // exit code goes here
    break;

  default:
    cout << "I didn't understand that" << endl;
}
```

Implementing an Example Calculator with the switch Statement

The following SwitchCalculator program uses the `switch` statement to implement a simple calculator:

```
// SwitchCalculator - use the switch statement to
//                    implement a calculator

#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // enter operand1 op operand2
    int  nOperand1;
    int  nOperand2;
    char cOperator;
    cout << "Enter 'value1 op value2'\n"
         << "where op is +, -, *, / or %:" << endl;
    cin >> nOperand1 >> cOperator >> nOperand2;

    // echo what the operator entered
    cout << nOperand1 << " "
         << cOperator << " "
         << nOperand2 << " = ";

    // now calculate the result; remember that the
    // user might enter something unexpected
    switch (cOperator)
    {
```

```
        case '+':
            cout << nOperand1 + nOperand2;
            break;
        case '-':
            cout << nOperand1 - nOperand2;
            break;
        case '*':
        case 'x':
        case 'X':
            cout << nOperand1 * nOperand2;
            break;
        case '/':
            cout << nOperand1 / nOperand2;
            break;
        case '%':
            cout << nOperand1 % nOperand2;
            break;
        default:
            // didn't understand the operator
            cout << " is not understood";
    }
    cout << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This program begins by prompting the user to enter `"value1 op value2"` where *op* is one of the common arithmetic operators `+`, `-`, `*`, `/` or `%`. The program then reads the variables `nOperand1`, `cOperator`, and `nOperand2`.

The program starts by echoing back to the user what it read from the keyboard. It follows this with the result of the calculation.

Echoing the input back to the user is always a good programming practice. It gives the user confirmation that the program read his input correctly.

The `switch` on `cOperator` differentiates between the operations that this calculator implements. For example, in the case that `cOperator` is `'+'`, the program reports the sum of `nOperand1` and `nOperand2`.

Because `'X'` is another common symbol for multiply, the program accepts `'*'`, `'X'`, and `'x'` all as synonyms for multiply using the case "fall through" feature. The program outputs an error message if `cOperator` doesn't match any of the known operators.

The output from a few sample runs appears as follows:

```
Enter 'value1 op value2'
where op is +, -, *, / or %:
22 x 6
22 x 6 = 132
Press any key to continue . . .
```

```
Enter 'value1 op value2'
where op is +, -, *, / or %:
22 / 6
22 / 6 = 3
Press any key to continue . . .
```

```
Enter 'value1 op value2'
where op is +, -, *, / or %:
22 % 6
22 % 6 = 4
Press any key to continue . . .
```

```
Enter 'value1 op value2'
where op is +, -, *, / or %:
22 $ 6
22 $ 6 =  is not understood
Press any key to continue . . .
```

Notice that the final run executes the default case of the `switch` statement since the character `'$'` did not match any of the cases.

Chapter 8

Debugging Your Programs, Part I

In This Chapter

- Avoiding introducing errors needlessly
- Creating test cases
- Peeking into the inner workings of your program
- Fixing and retesting your programs

You may have noticed that your programs often don't work the first time you run them. In fact, I have seldom, if ever, written a nontrivial C++ program that didn't have some type of error the first time I tried to execute it.

This leaves you with two alternatives: You can abandon a program that has an error, or you can find and fix the error. I assume that you want to take the latter approach. In this chapter, I first help you distinguish between types of errors and how to avoid errors in the first place. Then you get to find and eradicate two bugs that originally plagued the Conversion program in Chapter 3.

Identifying Types of Errors

Two types of errors exist — those that C++ can catch on its own and those that the compiler can't catch. Errors that C++ can catch are known as *compile-time* or *build-time errors*. Build-time errors are generally easier to fix because the compiler points you to the problem, if you can understand what the compiler's telling you. Sometimes the description of the problem isn't quite right (it's easy to confuse a compiler), but you start to understand better how the compiler thinks as you gain experience.

Errors that C++ can't catch don't show up until you try to execute the program during the process known as *unit testing*. During unit testing, you execute your program with a series of different inputs, trying to find inputs that make it crash. (You don't want your program to crash, of course, but better that you — rather than your user — find and correct these cases.)

The errors that you find by executing the program are known as *run-time errors*. Run-time errors are harder to find than build-time errors because you have no hint of what's gone wrong except for whatever errant output the program might generate.

The output isn't always so straightforward. For example, suppose that the program lost its way and began executing instructions that aren't even part of the program you wrote. (That happens a lot more often than you might think.) An errant program is like a train that's jumped the track — the program doesn't stop executing until it hits something really big. For example, the CPU may just happen to execute a divide by zero — this generates an alarm that the operating system intercepts and uses as an excuse to terminate your program.

An errant program is like a derailed train in another way — once the program starts heading down the wrong path, it *never* jumps back onto the track.

Not all run-time errors are quite so dramatic. Some errant programs stay on the tracks but generate the wrong output (almost universally known as "garbage output"). These are even harder to catch since the output may seem reasonable until you examine it closely.

In this chapter, you will debug a program that has both a compile time and a run-time error — not the "jump off the track and start executing randomly" variety but more of the generate garbage kind.

Avoiding Introducing Errors

The easiest and best way to fix errors is to avoid introducing them into your programs in the first place. Part of this is just a matter of experience, but adopting a clear and consistent programming style helps.

Coding with style

We humans have a limited amount of CPU power between our ears. We need to direct what CPU cycles we do have toward the act of creating a working program. We shouldn't get distracted by things like indentation.

This makes it important that you be consistent in how you name your variables, where you place open and close braces, how much you indent, and so on. This is called your coding style. Develop a style and stick to it. After a while your coding style will become second nature. You'll find that you can code your programs in less time and you can read the resulting programs