Making Everything Easier!™

6th Edition

C++ For Dummies®

Learn to:

- Understand C++ 2009 programming from the ground up
- Write your first program in Chapter 1
- Master classes and inheritance
- Sail right through streaming I/O

Stephen R. Davis
Coauthor of C# 2008 For Dummies

Chapter 1

Writing Your First C++ Program

In This Chapter

- Finding out about C++
- Installing Code::Blocks from the accompanying CD-ROM
- Creating your first C++ program
- Executing your program

Okay, so here we are: No one here but just you and me. Nothing left to do but get started. Might as well lay out a few fundamental concepts.

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.

Perhaps unfortunately for us, computers don't understand any reasonable human language — they don't speak English either. Okay, I know what you're going to say: "I've seen computers that could understand English." What you really saw was a computer executing a *program* that could meaningfully understand English.

Computers understand a language variously known as *computer language* or *machine language.* It's possible but extremely difficult for humans to speak machine language. Therefore, computers and humans have agreed to sort of meet in the middle, using intermediate languages such as C++. Humans can speak C++ (sort of), and C++ can be converted into machine language for the computer to understand.

Grasping C++ Concepts

A C++ program is a text file containing a sequence of C++ commands put together according to the laws of C++ grammar. This text file is known as the *source file* (probably because it's the source of all frustration). A C++ source file normally carries the extension `.CPP` just as a Microsoft Word file ends in `.DOC` or an MS-DOS (remember that?) batch file ends in `.BAT`.

The point of programming in C++ is to write a sequence of commands that can be converted into a machine-language program that actually *does* what we want done. This is called *compiling* and is the job of the compiler. The machine code that you wrote must be combined with some setup and tear-down instructions and some standard library routines in a process known as *linking* or *building*. The resulting *machine-executable* files carry the extension `.EXE` in Windows.

That sounds easy enough — so what's the big deal? Keep going.

To write a program, you need two specialized computer programs. One (an editor) is what you use to write your code as you build your `.CPP` source file. The other (a compiler) converts your source file into a machine-executable `.EXE` file that carries out your real-world commands (open spreadsheet, make rude noises, deflect incoming asteroids, whatever).

Nowadays, tool developers generally combine compiler and editor into a single package — a development *environment.* After you finish entering the commands that make up your program, you need only click a button to create the executable file.

Fortunately, there are public-domain C++ environments. We use one of them in this book — the Code::Blocks environment. This editor will work with a lot of different compilers, but a version of Code::Blocks combined with the GNU gcc compiler for 32-bit versions of Windows is included on the book's CD-ROM. You can download the most recent version of Code::Blocks from `www.codeblocks.org` or you can download a recent version that's been tested for compatibility with the programs in this book from the author's Web site at `www.stephendavis.com`.

You can download versions of the gcc compiler for the Mac or Linux from `www.gnu.org`.

Although Code::Blocks is public domain, you're encouraged to pay some small fee to support its further development. You don't *have* to pay to use Code::Blocks, but you can contribute to the cause if you like. See the Web site for details.

I have tested the programs in this book with Code::Blocks combined with gcc version 4.4; the programs should work with later versions as well. You can check out my Web site for a list of any problems that may arise with future versions of Code::Blocks, gcc, or Windows.

TECHNICAL STUFF

Code::Blocks is a full-fledged editor and development environment front end. Code::Blocks supports a multitude of different compilers including the gcc compiler included on the enclosed CD-ROM.

The Code::Blocks/gcc package generates Windows-compatible 32-bit programs, but it does not easily support creating programs that have the classic Windows look. I strongly recommend that you work through the examples in this book first to learn C++ *before* you tackle Windows development. C++ and Windows programming are two separate things and (for the sake of your sanity) should remain so in your mind.

Follow the steps in the next section to install Code::Blocks and build your first C++ program. This program's task is to convert a temperature value entered by the user from degrees Celsius to degrees Fahrenheit.

Installing Code::Blocks

The CD-ROM that accompanies this book includes the most recent version of the Code::Blocks environment at the time of this writing.

The Code::Blocks environment comes in an easy-to-install, compressed executable file. This executable file is contained in the `CodeBlocks` directory on the accompanying CD-ROM. Here's the rundown on installing the environment:

1. **Insert the CD into the CD-ROM drive.**
2. **When the CD interface appears with the License Agreement, click Accept.**
3. **Click the Installing Code::Blocks button on the left.**
4. **Click Open Directory.**
5. **Double-click the codeblocks_setup.exe file.**

   You can also choose Start➪Run and type ***x*:\codeblocks\setup** in the window that appears, where *x* is the letter designation for your CD-ROM drive (normally D).
6. **Depending on what version of Windows you're using, you may get the ubiquitous "An unidentified programs wants access to your computer" warning pop-up. If so, click Allow to get the installation ball rolling.**
7. **Click Next after closing all extraneous applications as you are warned in the Welcome dialog to the CodeBlocks Setup Wizard.**
8. **Read the End User Legal Agreement (commonly known as the EULA) and then click I Agree if you can live with its provisions.**

   It's not like you have much choice — the package really won't install itself if you don't accept. Assuming you *do* click OK, Code::Blocks opens a window showing the installation options. The default options are fine.

9. **Click the Next button.**

   The installation program asks where you want it to install Code::Blocks. This dialog box also shows you how much disk space the installation requires (and whether you have enough). The default is okay assuming you have enough disk space (if not, you'll have to delete one of your reruns of the Simpsons).

10. **Click Install.**

    Code::Blocks commences to copying a whole passel of files to your hard disk. Code::Blocks then asks "Do you want to run Code::Blocks now?"

11. **Click Yes to start Code::Blocks.**

    Code::Blocks now asks which compiler you intend to use. The default is GNU GCC Compiler, which is the proper selection.

12. **Click OK to select the GNU GCC compiler and start Code::Blocks.**

    Code::Blocks now wants to know which file associations you want to establish. The default is to allow Code::Blocks to open .CPP files. This option in fine unless you have another C++ compiler that you would rather have as the default.

13. **Select the Yes, Associate C++ Files with Code::Blocks option if you do not have another C++ compiler installed. Otherwise, select the No, Leave Everything as It Is option. Click OK.**

    You now need to make sure that the compiler options are set to enable all warnings and C++ 2009 features.

14. **From within Code::Blocks, choose Settings➪Compiler and Debugger. In the Compiler Flags tab, make sure that the Enable All Compiler Warnings is selected.**

15. **Select the Enable All Compiler Warnings option, as shown in Figure 1-1.**

    Start Code::Blocks. From within Code::Blocks, choose Settings➪Compiler and debugger. In the Compiler Flags tab, make sure that the Enable All Compiler Warnings is selected.

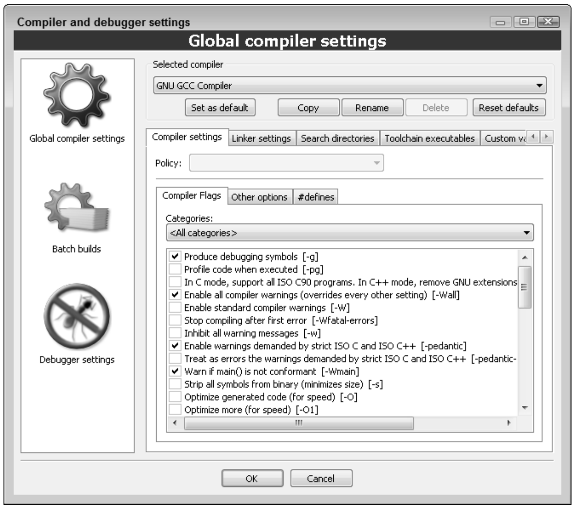
16. **Set the options to ensure C++ 2009 compliance.**

    The 2009 extensions are still considered a bit experimental as of this writing, so you need to tell gcc to enable these features. Click the Other Options tab and add the two lines `-std=c++0x` and `-Wc++0x-compat` as shown in Figure 1-2.

17. **Click OK.**

18. **Click Next in the Code::Blocks Setup window and then click Finish to complete the setup program.**

    The setup program exits.



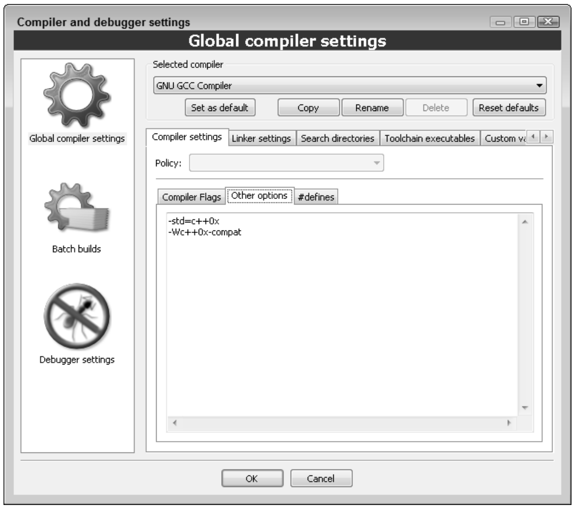
Selected compiler: GNU GCC Compiler

Policy:

Categories: <All categories>

- [✓] Produce debugging symbols [-g]
- [ ] Profile code when executed [-pg]
- [ ] In C mode, support all ISO C90 programs. In C++ mode, remove GNU extensions
- [✓] Enable all compiler warnings (overrides every other setting) [-Wall]
- [ ] Enable standard compiler warnings [-W]
- [ ] Stop compiling after first error [-Wfatal-errors]
- [ ] Inhibit all warning messages [-w]
- [✓] Enable warnings demanded by strict ISO C and ISO C++ [-pedantic]
- [ ] Treat as errors the warnings demanded by strict ISO C and ISO C++ [-pedantic-
- [✓] Warn if main() is not conformant [-Wmain]
- [ ] Strip all symbols from binary (minimizes size) [-s]
- [ ] Optimize generated code (for speed) [-O]
- [ ] Optimize more (for speed) [-O1]

Figure 1-1: Ensure that the Enable All Compiler Warnings is set.



Selected compiler: GNU GCC Compiler

Policy:

```
-std=c++0x
-Wc++0x-compat
```

Figure 1-2: Add these lines to enable the C++ 2009 features.

Creating Your First C++ Program

In this section, you create your first C++ program. You enter the C++ code into a file called `CONVERT.CPP` and then convert the C++ code into an executable program.

Creating a project

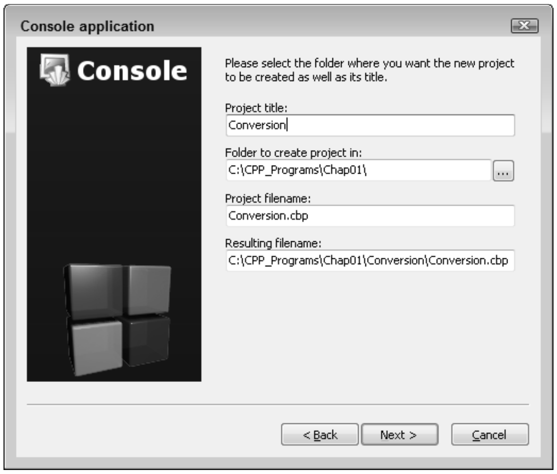
The first step to creating a C++ program is to create what is known as a project. A *project* tells Code::Blocks the names of the .CPP source files to include and what type of program to create. Most of the programs in the book will consist of a single source file and will be command-line style:

1. **Choose Start➪Programs➪CodeBlocks➪CodeBlocks to start up the CodeBlocks tool.**
2. **From within Code::Blocks, choose File➪New➪Project.**
3. **Select the Console Application icon and then click Go.**
4. **Select C++ as the language you want to use from the next window. Click Next.**

   Code::Blocks and gcc also support plain ol' C programs.
5. **Select the Console Application icon and then click Go.**
6. **In the Folder to Build Project In field, navigate to the subdirectory where you want your program built.**

   I have divided the programs in this book by chapter, so I created the folder `C:\CPP_Programs\Chap01` using Windows Explorer and then selected it from the file menu.
7. **In the Project Title field, type the name of the project, in this case** Conversion**.**

   The resulting screen is shown in Figure 1-3.



Please select the folder where you want the new project to be created as well as its title.

Project title: Conversion

Folder to create project in: C:\CPP_Programs\Chap01\

Project filename: Conversion.cbp

Resulting filename: C:\CPP_Programs\Chap01\Conversion\Conversion.cbp

Figure 1-3: I created the project Conversion for the first program.

8. **Click Next.**

   The next window gives you the option of creating an application for testing or the final version. The default is fine.

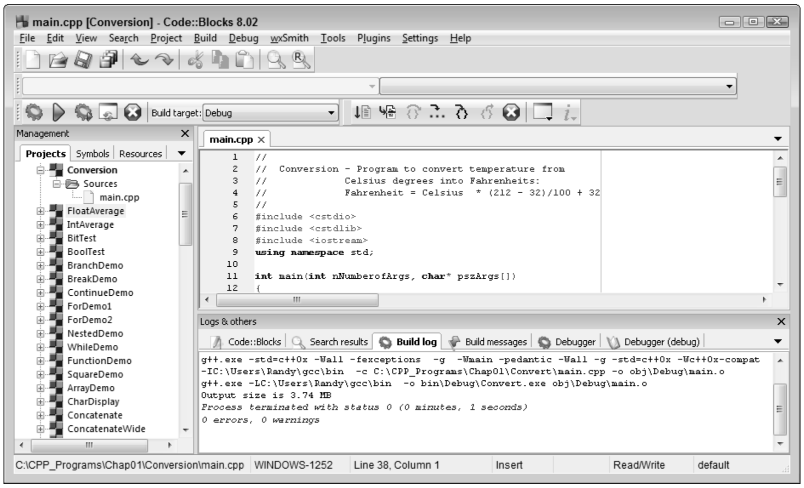
9. **Click Finish to create the Conversion project.**

Entering the C++ code

The Conversion project that Code::Blocks creates consists of a single, default main.cpp file that does nothing. The next step is to enter our program:

1. **In the Management window on the left, double-click main, which is under Sources, which is under Conversion.**

   Code::Blocks opens the empty main.cpp program that it created in the code editor, as shown in Figure 1-4. (Figure 1-4 shows all the projects from the entire book as well.)



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

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

```
g++.exe -std=c++0x -Wall -fexceptions  -g  -Wmain -pedantic -Wall -g -std=c++0x -Wc++0x-compat
-IC:\Users\Randy\gcc\bin  -c C:\CPP_Programs\Chap01\Convert\main.cpp -o obj\Debug\main.o
g++.exe -LC:\Users\Randy\gcc\bin  -o bin\Debug\Convert.exe obj\Debug\main.o
Output size is 3.74 MB
Process terminated with status 0 (0 minutes, 1 seconds)
0 errors, 0 warnings
```

Figure 1-4: The Management window displays a directory structure for all available programs.

2. **Edit main.cpp with the following program exactly as written.**

   Don't worry too much about indentation or spacing — it isn't critical whether a given line is indented two or three spaces, or whether there are one or two spaces between two words. C++ is case sensitive, however, so you need to make sure everything is lowercase.

   TIP

   You can cheat by using the files contained on the enclosed CD-ROM as described in the next section "Cheating."

```
//
//  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;

  // calculate conversion factor for Celsius
  // to Fahrenheit
  int factor;
  factor = 212 - 32;

  // use conversion factor to convert Celsius
  // into Fahrenheit values
  int fahrenheit;
  fahrenheit = factor * celsius/100 + 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;
}
```

3. **Choose File➪Save to save the source file.**

   I know that it may not seem all that exciting, but you've just created your first C++ program!

Cheating

All the programs in the book are included on the enclosed CD-ROM along with the project files to build them. You can use these files in two ways: one way is to go through all the steps to create the program by hand first but copy and paste from the sources on the CD-ROM into your program if you get into trouble (or your fingers start cramping). This is the preferred technique.

Alternatively you can use the following procedure to copy all the programs to your hard disk at once:

1. **Copy all the sources from the CD-ROM to the hard disk.**

   This copies all the source code from the book along with the project files to build those programs.

2. **Double-click `AllPrograms.workspace` in C:\CPP_Programs.**

   A workspace is a single file that references one or more projects. The `AllPrograms.workspace` file contains references to all the projects defined in the book.

3. **Right-click the Conversion project in the Management window on the left. Choose Activate Project from the context-sensitive menu that appears.**

   Code::Blocks turns the Conversion label bold to verify that this is the program you are working with right now.

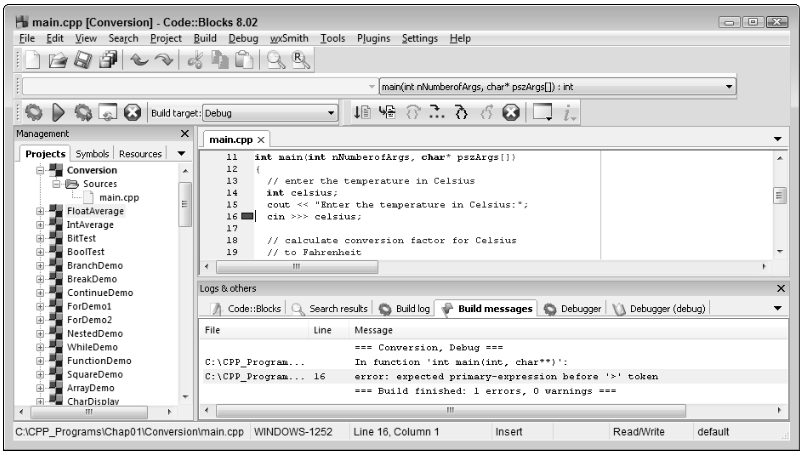
Building your program

After you've saved your C++ source file to disk, it's time to generate the executable machine instructions.

To build your Conversion program, you choose Build➪Build from the menu or press Ctrl-F9. Almost immediately, Code::Blocks takes off, compiling your program with gusto. If all goes well, the happy result of *0 errors, 0 warnings* appears in the lower-right window.

Code::Blocks generates a message if it finds any type of error in your C++ program — and coding errors are about as common as ice cubes in Alaska. You'll undoubtedly encounter numerous warnings and error messages, probably even when entering the simple `Conversion.cpp`. To demonstrate the error-reporting process, let's change Line 16 from `cin >> celsius;` to `cin >>> celsius;`.

This seems an innocent enough offense — forgivable to you and me perhaps, but not to C++. Choose Build➪Build to start the compile and build process. Code::Blocks almost immediately places a red square next to the erroneous line as shown in Figure 1-5. The message in the Build Message tab is a rather cryptic `error: expected primary-expression before '>' token`. To get rid of the message, remove the extra > and recompile.


main.cpp [Conversion] - Code::Blocks 8.02
File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help
main(int nNumberofArgs, char* pszArgs[]) : int
Build target: Debug
Management
Projects Symbols Resources
Conversion
Sources
main.cpp
FloatAverage
IntAverage
BitTest
BoolTest
BranchDemo
BreakDemo
ContinueDemo
ForDemo1
ForDemo2
NestedDemo
WhileDemo
FunctionDemo
SquareDemo
ArrayDemo
CharDisplay
main.cpp
Logs & others
Code::Blocks Search results Build log Build messages Debugger Debugger (debug)

```
11  int main(int nNumberofArgs, char* pszArgs[])
12  {
13    // enter the temperature in Celsius
14    int celsius;
15    cout << "Enter the temperature in Celsius:";
16    cin >>> celsius;
17
18    // calculate conversion factor for Celsius
19    // to Fahrenheit
```

```
File                  Line  Message
                            === Conversion, Debug ===
C:\CPP_Program...           In function 'int main(int, char**)':
C:\CPP_Program...     16    error: expected primary-expression before '>' token
                            === Build finished: 1 errors, 0 warnings ===
```

```
C:\CPP_Programs\Chap01\Conversion\main.cpp  WINDOWS-1252  Line 16, Column 1  Insert  Read/Write  default
```

Figure 1-5: Code::Blocks flags the source of errors quickly.

TIP

You probably consider the error message generated by the example a little cryptic but give it time — you've been programming for only about 30 minutes now. Over time you'll come to understand the error messages generated by Code::Blocks and gcc much better.

WARNING!

Code::Blocks was able to point directly at the error this time but it isn't always that good. Sometimes it doesn't notice the error until the next line or the one after that, so if the line flagged with the error looks okay, start looking at its predecessor to see if the error is there.

Executing Your Program

It's now time to execute your new creation . . . that is, to run your program. You will run the `CONVERT.EXE` program file and give it input to see how well it works.

To execute the Conversion program, choose Build➪Build and Run or press F9. This rebuilds the program if anything has changed and executes the program if the build is successful.

A window opens immediately, requesting a temperature in Celsius. Enter a known temperature, such as 100 degrees. After you press Enter, the program returns with the equivalent temperature of 212 degrees Fahrenheit as follows:

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

The message `Press any key to continue...` gives you the opportunity to read what you've entered before it goes away. Press Enter, and the window (along with its contents) disappears. Congratulations! You just entered, built, and executed your first C++ program.

Notice that Code::Blocks is not truly intended for developing Windows programs. In theory, you can write a Windows application by using Code::Blocks, but it isn't easy. (Building windowed applications is *so* much easier in Visual Studio.NET.)

Windows programs show the user a visually oriented output, all nicely arranged in onscreen windows. Conversion.exe is a 32-bit program that executes *under* Windows, but it's not a Windows program in the visual sense.

If you don't know what *32-bit program* means, don't worry about it. As I said, this book isn't about writing Windows programs. The C++ programs you write in this book have a *command line interface* executing within an MS-DOS box.

Budding Windows programmers shouldn't despair — you didn't waste your money. Learning C++ is a prerequisite to writing Windows programs. I think that they should be mastered separately: C++ first, Windows second.

Reviewing the Annotated Program

Entering data in someone else's program is about as exciting as watching someone else drive a car. You really need to get behind the wheel itself. Programs are a bit like cars as well. All cars are basically the same with small differences and additions — okay, French cars are a lot different than other cars, but the point is still valid. Cars follow the same basic pattern — steering wheel in front of you, seat below you, roof above you, and stuff like that.

Similarly, all C++ programs follow a common pattern. This pattern is already present in this very first program. We can review the Conversion program by looking for the elements that are common to all programs.

Examining the framework for all C++ programs

Every C++ program you write for this book uses the same basic framework, which looks a lot like this:

```
//
//  Template - provides a template to be used as the
//             starting point
//
// the following include files define the majority of
// functions that any given program will need
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // your C++ code starts here

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

Without going into all the boring details, execution begins with the code contained in the open and closed braces immediately following the line beginning `main()`.

I've copied this code into a file called `Template.cpp` located in the main `CPP_Programs` folder on the enclosed CD-ROM.

Clarifying source code with comments

The first few lines in the Conversion program appear to be freeform text. Either this code was meant for human eyes or C++ is a lot smarter than I give it credit for. These first six lines are known as comments. *Comments* are the programmer's explanation of what she is doing or thinking when writing a particular code segment. The compiler ignores comments. Programmers (*good* programmers, anyway) don't.

A C++ comment begins with a double slash (`//`) and ends with a newline. You can put any character you want in a comment. A comment may be as long as you want, but it's customary to keep comment lines to no more than 80 characters across. Back in the old days — "old" is relative here — screens were limited to 80 characters in width. Some printers still default to 80 characters across when printing text. These days, keeping a single line to fewer than 80 characters is just a good practical idea (easier to read; less likely to cause eyestrain; the usual).

A newline was known as a *carriage return* back in the days of typewriters — when the act of entering characters into a machine was called *typing* and not *keyboarding*. A *newline* is the character that terminates a command line.

C++ allows a second form of comment in which everything appearing after a `/*` and before a `*/` is ignored; however, this form of comment isn't normally used in C++ anymore. (Later in this book, I describe the one case in which this type of comment is applied.)

It may seem odd to have a command in C++ (or any other programming language) that's specifically ignored by the computer. However, all computer languages have some version of the comment. It's critical that the programmer explain what was going through her mind when she wrote the code. A programmer's thoughts may not be obvious to the next colleague who tries to use or modify her program. In fact, the programmer herself may forget what her program meant if she looks at it months after writing the original code and has left no clue.

Basing programs on C++ statements

All C++ programs are based on what are known as C++ statements. This section reviews the statements that make up the program framework used by the Conversion program.

A statement is a single set of commands. Almost all C++ statements other than comments end in a semicolon. (You see one other exception in Chapter 10). Program execution begins with the first C++ statement after the open brace and continues through the listing, one statement at a time.

As you look through the program, you can see that spaces, tabs, and newlines appear throughout the program. In fact, I place a newline after every statement in this program. These characters are collectively known as *whitespace* because you can't see them on the monitor.

You may add whitespace anywhere you like in your program to enhance readability — except in the middle of a word:

```
See wha

t I mean?
```

Although C++ may ignore whitespace, it doesn't ignore case. In fact, C++ is case sensitive to the point of obsession. The variable `fullspeed` and the variable `FullSpeed` have nothing to do with each other. The command `int` is completely understandable, but C++ has no idea what `INT` means. See what I mean about fast but stupid compilers?

Writing declarations

The line `int nCelsius;` is a declaration statement. A *declaration* is a statement that defines a variable. A *variable* is a "holding tank" for a value of some type. A variable contains a *value,* such as a number or a character.

The term *variable* stems from algebra formulas of the following type:

```
x = 10
y = 3 * x
```

In the second expression, `y` is set equal to 3 times x, but what is `x`? The variable `x` acts as a holding tank for a value. In this case, the value of `x` is 10, but we could have just as well set the value of `x` to 20 or 30 or –1. The second formula makes sense no matter what the value of `x` is.

In algebra, you're allowed but not required to begin with a statement such as `x = 10`. In C++, the programmer must define the variable `x` before she can use it.

In C++, a variable has a type and a name. The variable defined on line 11 is called `celsius` and declared to hold an integer. (Why they couldn't have just said *integer* instead of *int,* I'll never know. It's just one of those things you learn to live with.)

The name of a variable has no particular significance to C++. A variable must begin with the letters *A* through *Z,* the letters *a* through *z,* or an underscore (_). All subsequent characters must be a letter, a digit 0 through 9, or an underscore. Variable names can be as long as you want to make them.

It's convention that variable names begin with a lowercase letter. Each new word *within* a variable begins with a capital letter, as in `myVariable`.

Try to make variable names short but descriptive. Avoid names such as `x` because x has no particular meaning. A variable name such as `lengthOfLineSegment` is much more descriptive.

Generating output

The lines beginning with `cout` and `cin` are known as input/output statements, often contracted to I/O statements. (Like all engineers, programmers love contractions and acronyms.)

The first I/O statement says "Output the phrase *Enter the temperature in Celsius* to *cout*" (pronounced "see-out"). `cout` is the name of the standard C++ output device. In this case, the standard C++ output device is your monitor.

The next line is exactly the opposite. It says, in effect, "Extract a value from the C++ input device and store it in the integer variable `celsius`." The C++ input device is normally the keyboard. What we have here is the C++ analog to the algebra formula `x = 10` just mentioned. For the remainder of the program, the value of `celsius` is whatever the user enters there.

Calculating Expressions

All but the most basic programs perform calculations of one type or another. In C++, an *expression* is a statement that performs a calculation. Said another way, an expression is a statement that *has a value.* An *operator* is a command that generates a value.

For example, in the Conversion example program — specifically in the two lines marked as a `calculation expression` — the program declares a variable *factor* and then assigns it the value resulting from a calculation. This particular command calculates the difference of 212 and 32; the operator is the minus sign (–), and the expression is `212-32`.

Storing the results of an expression

The spoken language can be very ambiguous. The term *equals* is one of those ambiguities. The word *equals* can mean that two things have the same value as in "a dollar equals one hundred cents." Equals can also imply assignment, as in math when you say that "y equals 3 times x."

To avoid ambiguity, C++ programmers call = the *assignment operator,* which says (in effect), "Store the results of the expression to the right of the `equal` sign in the variable to the left." Programmers say that "`factor` is assigned the value 212 minus 32." For short, you can say "factor gets 212 minus 32."

WARNING!

Never say "`factor` is *equal to* 212 minus 32." You'll hear this from some lazy types, but you and I know better.

Examining the remainder of Conversion

The second expression in the Conversion program presents a slightly more complicated expression than the first. This expression uses the same mathematical symbols: * for multiplication, / for division, and + for addition. In this case, however, the calculation is performed on variables and not simply on constants.

The value contained in the variable called `factor` (which was calculated as the results of 212 – 32, by the way) is multiplied by the value contained in `celsius` (which was input from the keyboard). The result is divided by 100 and summed with 32. The result of the total expression is assigned to the integer variable `fahrenheit`.

The final two commands output the string `Fahrenheit value is:` to the display, followed by the value of `fahrenheit` — and all so fast that the user scarcely knows it's going on.

Chapter 2

Declaring Variables Constantly

In This Chapter

- ▶ Declaring variables
- ▶ Declaring different types of variables
- ▶ Using floating-point variables
- ▶ Declaring and using other variable types

The most fundamental of all concepts in C++ is the *variable* — a variable is like a small box. You can store things in the box for later use, particularly numbers. The concept of a variable is borrowed from mathematics. A statement such as

```
x = 1
```

stores the value 1 in the variable x. From that point forward, the mathematician can use the variable x in place of the constant 1 — until he changes the value of x to something else.

Variables work the same way in C++. You can make the assignment

```
x = 1;
```

From that point forward in the execution of the program, until the value of x is changed, the *value of* x is 1. References to x are replaced by the value 1. In this chapter, you will find out how to declare and initialize variables in C++ programs. You will also see the different types of variables that C++ defines and when to use each.

Declaring Variables

A mathematician might write something like the following:

```
 (x + 2) = y / 2
x + 4 = y
solve for x and y
```

Any reader who's had algebra realizes right off that the mathematician has introduced the variables `x` and `y`. But C++ isn't that smart. (Computers may be fast, but they're stupid.)

You have to announce each variable to C++ before you can use it. You have to say something soothing like this:

```
int x;
x = 10;

int y;
y = 5;
```

These lines of code *declare* that a variable `x` exists, is of type `int`, and has the value 10; and that a variable `y` of type `int` also exists with the value 5. (The next section discusses variable types.) You can declare variables (almost) anywhere you want in your program — as long as you *declare the variable before you use it.*

Declaring Different Types of Variables

If you're on friendly terms with math (and who isn't?), you probably think of a variable in mathematics as an amorphous box capable of holding whatever you might choose to store in it. You might easily write something like the following:

```
x = 1
x = 2.3
x = "this is a sentence"
```

Alas, C++ is not that flexible. (On the other hand, C++ can do things that people can't do, such as add a billion numbers or so in a second, so let's not get too uppity.) To C++, there are different types of variables just as there are different types of storage bins. Some storage bins are so small that they can handle only a single number. It takes a larger bin to handle a sentence.

Some computer languages try harder to accommodate the programmer by allowing her to place different types of data in the same variable. These languages are called *weakly typed* languages. C++ is a *strongly typed* language — it requires the programmer to specifically declare each variable along with its exact type.

The variable type int is the C++ equivalent of an *integer* — a number that has no fractional part. (Integers are also known as *counting numbers* or *whole numbers.*)

Integers are great for most calculations. I made it through most of elementary school with integers. It isn't until I turned 11 or so that my teachers started mucking up the waters with fractions. The same is true in C++: More than 90 percent of all variables in C++ are declared to be of type int.

Unfortunately, int variables aren't adapted to every problem. For example, if you worked through the temperature-conversion program in Chapter 1, you might have noticed that the program has a potential problem — it can calculate temperatures to the nearest degree. No fractions of a degree are allowed. This integer limitation wouldn't affect daily use because it isn't likely that someone (other than a meteorologist) would get all excited about being off a fraction of a degree. There are plenty of cases, however, where this isn't the case — for example, you wouldn't want to come up a half mile short of the runway on your next airplane trip due to a navigational round-off.

Reviewing the limitations of integers in C++

The int variable type is the C++ version of an integer. int variables suffer the same limitations as their counting-number integer equivalents in math do.

Integer round-off

Lopping off the fractional part of a number is called *truncation*. Consider the problem of calculating the average of three numbers. Given three int variables — nValue1, nValue2, and nValue3 — an equation for calculating the average is

```
int nAverage; int nValue1; int nValue2; int nValue3;
nAverage = (nValue1 + nValue2 + nValue3) / 3;
```

Because all three values are integers, the sum is assumed to be an integer. Given the values 1, 2, and 2, the sum is 5. Divide that by 3, and you get 1⅔, or 1.666. C++ uses slightly different rules: Given that all three variables nValue1, nValue2, and nValue3 are integers, the sum is also assumed to be an integer. The result of the division of one integer by another integer is also an integer. Thus, the resulting value of nAverage is the unreasonable but logical value of 1.

The problem is much worse in the following mathematically equivalent formulation:

```
int nAverage; int nValue1; int nValue2; int nValue3;
nAverage = nValue1/3 + nValue2/3 + nValue3/3;
```

Plugging in the same 1, 2, and 2 values, the resulting value of `nAverage` is 0 (talk about logical but unreasonable). To see how this can occur, consider that $^{1}/_{3}$ truncates to 0, $^{2}/_{3}$ truncates to 0, and $^{2}/_{3}$ truncates to 0. The sum of 0, 0, and 0 is 0. You can see that integer truncation can be completely unacceptable.

Limited range

A second problem with the `int` variable type is its limited range. A normal `int` variable can store a maximum value of 2,147,483,647 and a minimum value of –2,147,483,648 — roughly from positive 2 billion to negative 2 billion, for a total range of about 4 billion.

Two billion is a very large number: plenty big enough for most uses. But it's not large enough for some applications, including computer technology. In fact, your computer probably executes faster than 2 gigahertz, depending on how old your computer is. (*Giga* is the prefix meaning billion.) A single strand of communications fiber — the kind that's been strung back and forth from one end of the country to the other — can handle way more than 2 billion bits per second.

Solving the truncation problem

The limitations of `int` variables can be unacceptable in some applications. Fortunately, C++ understands decimal numbers that have a fractional part. (Mathematicians also call those *real numbers.*) Decimal numbers avoid many of the limitations of `int` type integers. To C++ all decimal numbers have a fractional part even if that fractional part is 0. In C++, the number 1.0 is just as much a decimal number as 1.5. The equivalent integer is written simply as 1. Decimal numbers can also be negative, such as –2.3.

When you declare variables in C++ that are decimal numbers, you identify them as floating-point or simply `float` variables. The term *floating-point* means the decimal point is allowed to float back and forth, identifying as many decimal places as necessary to express the value. Floating-point variables are declared in the same way as `int` variables:

```
float fValue1;
```

Once declared, you cannot change the type of a variable. `fValue1` is now a `float` and will be a `float` for the remainder of the program. To see how floating-point numbers fix the truncation problem inherent with integers, convert all the `int` variables to `float`. Here's what you get:

```
float fValue;
fValue = 1.0/3.0 + 2.0/3.0 + 2.0/3.0;
```

is equivalent to

```
fValue = 0.333... + 0.666... + 0.666...;
```

which results in the value

```
fValue = 1.666...;
```

I have written the value `1.6666 . . .` as if the number of trailing 6s goes on forever. This is not necessarily the case. A `float` variable has a limit to the number of digits of accuracy, but it's a lot more than I can keep track of.

A constant that has a decimal point is assumed to be a floating-point value. However, the default type for a floating-point constant is something known as a double precision, which in C++ is called simply `double`, as we'll see in the next section.

The programs `IntAverage` and `FloatAverage` are available on the enclosed CD in the `CPP_Programs\Chap02` directory to demonstrate the round-off error inherent in integer variables.

Looking at the limits of floating-point numbers

Although floating-point variables can solve many calculation problems such as truncation, they have some limitations themselves — the reverse of those associated with integer variables. Floating-point variables can't be used to count things, are more difficult for the computer to handle, and also suffer from round-off error (though not nearly to the same degree as `int` variables).

Counting

You cannot use floating-point variables in applications where counting is important. This includes C++ constructs that count. C++ can't verify which whole number value is meant by a given floating-point number.

For example, it's clear to you and me that 1.0 is 1 but not so clear to C++. What about 0.9 or 1.1? Should these also be considered as 1? C++ simply avoids the problem by insisting on using `int` values when counting is involved.

Calculation speed

Historically, a computer processor can process integer arithmetic quicker than it can floating-point arithmetic. Thus, while a processor can add 1 million integer numbers in a given amount of time, the same processor may be able to perform only 200,000 floating-point calculations during the same period.

Calculation speed is becoming less of a problem as microprocessors get faster. In addition, today's general-purpose microprocessors include special floating-point circuitry on board to increase the performance of these operations. However, arithmetic on integer values is just a heck of a lot easier and faster than performing the same operation on floating-point values.

Loss of accuracy

Floating-point `float` variables have a precision of about 6 digits, and an extra-economy size, double-strength version of `float` known as a `double` can handle about 13 significant digits. This can cause round-off problems as well.

Consider that $^1/_3$ is expressed as 0.333 . . . in a continuing sequence. The concept of an infinite series makes sense in math but not to a computer because it has a finite accuracy. The `FloatAverage` program outputs 1.66667 as the average 1, 2, and 2 — that's a lot better than the 0 output by the `IntAverage` version but not even close to an infinite sequence.

C++ can correct for round-off error in a lot of cases. For example, on output, C++ can sometimes determine that the user really meant 1 instead of 0.999999. In other cases, even C++ cannot correct for round-off error.

Not-so-limited range

Although the `double` data type has a range much larger than that of an integer, it's still limited. The maximum value for an `int` is a skosh more than 2 billion. The maximum value of a `double` variable is roughly 10 to the 38th power. That's 1 followed by 38 zeroes; it eats 2 billion for breakfast. (It's even more than the national debt, at least at the time of this writing.)

WARNING!

Remember, only the first 13 digits or so have any meaning; the remaining 25 digits are noise having succumbed to floating-point round-off error.

Declaring Variable Types

So far in this chapter, I have been trumpeting that variables must be declared and that they must be assigned a type. Fortunately (ta-dah!), C++ provides a number of variable types. See Table 2-1 for a list of variables, their advantages, and limitations.

Table 2-1 Common C++ Variable Types

Variable	*Defining a Constant*	*What It Is*
`int`	`1`	A simple counting number, either positive or negative.
`short int`	`---`	A potentially smaller version of `int`. It uses less memory but has a smaller range.
`long int`	`10L`	A potentially larger version of `int`. There is no difference between `long` and `int` with gcc.
`long long int`	`10LL`	A potentially even larger version of `int`.
`float`	`1.0F`	A single precision real number. This smaller version takes less memory than a `double` but has less accuracy and a smaller range.
`double`	`1.0`	A standard floating-point variable.
`long double`	`---`	A potentially larger floating-point number. On the PC, `long double` is used for the native size of the 80x86 floating-point processor, which is 80 bits.
`char`	`'c'`	A single `char` variable stores a single alphabetic or digital character. Not suitable for arithmetic.
`wchar_t`	`L'c'`	A larger character capable or storing symbols with larger character sets like Chinese.
`char string`	`"this is a string"`	A string of characters forms a sentence or phrase.
`bool`	`true`	The only other value is false. No, I mean, it's *really* false. Logically false. Not *false* as in fake or ersatz or . . . never mind.

C++ '09 NEW

The `long long int` and `long double` were officially introduced with C++ '09.

The integer types come in both signed and unsigned versions. Signed is always the default (for everything except `char` and `wchar_t`). The unsigned version is created by adding the keyword `unsigned` in front of the type in the declaration. The unsigned constants include a *U* or *u* in their type designation. Thus, the following declares an `unsigned int` variable and assigns it the value 10:

```
unsigned int uVariable;
uVariable = 10U;
```

The following statement declares the two variables `lVariable1` and `lVariable2` as type `long int` and sets them equal to the value 1, while `dVariable` is a double set to the value 1.0. Notice in the declaration of `lVariable2` that the `int` is assumed and can be left off:

```
// declare two long int variables and set them to 1
long int lVariable1
long lVariable2;     // int is assumed
lVariable1 = lVariable2 = 1;
// declare a variable of type double and set it to 1.0
double dVariable; dVariable = 1.0;
```

TIP

You can declare a variable and initialize it in the same statement:

```
int nVariable = 1;  // declare a variable and
                    // initialize it to 1
```

A `char` variable can hold a single character; a character string (which isn't really a variable type but works like one for most purposes) holds a string of characters. Thus, `'C'` is a `char` that contains the character C, whereas `"C"` is a string with one character in it. A rough analogy is that a `'C'` corresponds to a nail in your hand, whereas `"C"` corresponds to a nail gun with one nail left in the magazine. (Chapter 9 describes strings in detail.)

WARNING!

If an application requires a string, you've gotta provide one, even if the string contains only a single character. Providing nothing but the character just won't do the job.

Types of constants

A *constant value* is an explicit number or character (such as 1, 0.5, or 'c') that doesn't change. As with variables, every constant has a type. In an expression such as `n = 1;` the constant value 1 is an `int`. To make 1 a `long` integer, write the statement as `n = 1L;`. The analogy is as follows: 1 represents a pickup truck with one ball in it, whereas 1L is a dump truck also with one ball. The number of balls is the same in both cases, but the capacity of one of the containers is much larger.

Following the int to long comparison, 1.0 represents the value 1 but in a floating-point container. Notice, however, that the default for floating-point constants is double. Thus, 1.0 is a double number and not a float.

The constant values true and false are of type bool. In keeping with C++'s attention to case, true is a constant but TRUE has no meaning.

A variable can be declared constant when it is created via the keyword const:

```
const double PI = 3.14159; // declare a constant variable
```

A const variable must be initialized with a value when it is declared, and its value cannot be changed by any future statement.

TIP

Variables declared const don't have to be named with all capitals, but by convention they often are. This is just a hint to the reader that this so-called variable is, in fact, not.

I admit that it may seem odd to declare a variable and then say that it can't change. Why bother? Largely because carefully named const variables can make a program a lot easier to understand. Consider the following two equivalent expressions:

```
double dC = 6.28318 * dR;  // what does this mean?
double dCircumference = 2 * PI * dRadius; // this is a
                           // lot easier to understand
```

It should be a lot clearer to the reader of this code that the second expression is multiplying the radius of something by 2π to calculate the circumference.

Range of Numeric Types

It may seem odd but the C++ standard doesn't say exactly how big a number each of the data types can accommodate. The standard speaks only to the relative size of each data type. For example, it says that the maximum long int is at least as large as the maximum int.

The authors of C++ weren't trying to be mysterious. They merely wanted to allow the compiler to implement the absolute fastest code possible for the base machine. The standard was designed to work for all different types of processors running different operating systems.

However, it is useful to know the limits for your particular implementation. Table 2-2 shows the size of each number type on a Windows PC using the Code::Blocks/gcc compiler that comes on the enclosed CD-ROM.

Table 2-2 Range of Numeric Types in Code::Block/gcc

Variable	*Size (bytes)*	*Accuracy*	*Range*
`short`	2	exact	–32768 to 32767
`int`	4	exact	–2,147,483,648 to 2,147,483,647
`long`	4	exact	–2,147,483,648 to 2,147,483,647
`long long int`	8	exact	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4	7 digits	$\pm 3.4028 \times 10^{\pm 38}$
`double`	8	16 digits	$\pm 1.7977 \times 10^{\pm 308}$
`long double`	12	19 digits	$\pm 1.1897 \times 10^{\pm 4932}$

Attempting to calculate a number that's beyond the range of its type is known as an *overflow.* The C++ standard generally leaves the results of an overflow as undefined. That's another way that the definers of C++ remained flexible.

TIP

On the PC, a floating-point overflow results in an exception, which if not handled will cause your program to crash. (I don't discuss exceptions until Chapter 24.) As bad as that sounds, an integer overflow is worse — C++ silently generates an incorrect value without complaint.

C++ collision with filenames

Windows uses the backslash character to separate folder names in the path to a file. (This is a remnant of MS-DOS that Windows has not been able to shake.) Thus, `Root\FolderA\File` represents `File` within `FolderA`, which is a subdirectory of `Root`.

Unfortunately, MS-DOS's use of the backslash conflicts with the use of the backslash to indicate an escape character in C++. The character `\\` is a backslash in C++. The MS-DOS path `Root\FolderA\File` is represented in C++ as the string "`Root\\FolderA\\File`".

Special characters

You can store any printable character you want in a `char` or `string` variable. You can also store a set of nonprintable characters that are used as character constants. See Table 2-3 for a description of these important nonprintable characters.

Table 2-3 Special Characters

Character Constant	***What It Is***
`'\n'`	newline
`'\t'`	tab
`'\040'`	The character whose value is 40 in octal (see Chapter 4 for a discussion of number systems)
`'\x20'`	The character whose value is 20 in hexadecimal (this is the same as `'\040'`)
`'\0'`	null (i.e., the character whose value is 0)
`'\\'`	backslash

You have already seen the newline character at the end of strings. This character breaks a string and puts the parts on separate lines. A newline character may appear anywhere within a string. For example:

```
"This is line 1\nThis is line 2"
```

appears on the output as

```
This is line 1
This is line 2
```

Similarly, the `\t` tab character moves output to the next tab position. (This position can vary, depending on the type of computer you're using to run the program.)

The numerical forms allow you to specify any nonprinting character that you like, but results may vary. The character represented by `0xFB`, for example, depends on the font and the character set (and may not be a legal character at all).

Because the backslash character is used to signify special characters, a character pair for the backslash itself is required. The character pair \\ represents the backslash.

Wide Loads on Char Highway

The standard `char` variable is a scant 1 byte wide and can handle only 255 different characters. This is plenty enough for European languages but not big enough to handle symbol-based languages such as kanji.

Several standards have arisen to extend the character set to handle the demands of these languages. UTF-8 uses a mixture of 8-, 16-, and 32-bit characters to implement almost every kanji or hieroglyph you can think of but still remain compatible with simple 8-bit ASCII. UTF-16 uses a mixture of 16- and 32-bit characters to achieve an expanded character set, and UTF-32 uses 32 bits for all characters.

TIP

UTF stands for Unicode Transformation Format, from which it gets the common nickname Unicode.

C++ '09 NEW

Table 2-4 describes the different character types supported by C++. At first, C++ tried to get by with a vaguely defined wide character type, `wchar_t`. This type was intended to be the wide character type native to the application program's environment. C++ '09 introduced specific types for UTF-16 and UTF-32.

TIP

UTF-16 is the standard encoding for Windows and .NET applications. The `wchar_t` type refers to UTF-16 in the Code::Blocks/gcc compiler included on the CD-ROM.

Table 2-4 The C++ Character Types

Variable	*Example*	*What It Is*
`char`	`'c'`	ASCII character
`wchar_t`	`L'c'`	Character in wide format
`char_16t`	`u'c'`	UTF-16 character
`char_32t`	`U'c'`	UTF-32 character

Any of the character types in Table 2-4 can be combined into strings as well:

```
wchar_t* wideString = L"this is a wide string";
```

(Ignore the asterisk for now. I have a lot to say about its meaning in Chapter 8.)

Are These Calculations Really Logical?

C++ provides a logical variable called `bool`. The type `bool` comes from *Boole,* the last name of the inventor of the logical calculus. A Boolean variable has two values: `true` and `false`.

TIP

There are actually calculations that result in the value `bool`. For example, `"x is equal to y"` is either `true` or `false`.

Mixed Mode Expressions

C++ allows you to mix variable types in a single expression. That is, you are allowed to add an integer with a `double` precision floating-point value. In the following expression, for example, `nValue1` is allowed to be an `int`:

```
// in the following expression the value of nValue1
// is converted into a double before performing the
// assignment
int nValue1 = 1;
nValue1 + 1.0;
```

An expression in which the two operands are not the same type is called a *mixed-mode expression.* Mixed-mode expressions generate a value whose type is equal to the more capable of the two operands. In this case, `nValue1` is converted to a `double` before the calculation proceeds. Similarly, an expression of one type may be assigned to a variable of a different type, as in the following statement:

```
// in the following assignment, the whole
// number part of fVariable is stored into nVariable
double dVariable = 1.0;
int nVariable;
nVariable = dVariable;
```

WARNING!

You can lose precision or range if the variable on the left side of the assignment is smaller. In the preceding example, C++ truncates the value of `dVariable` before storing it in `nVariable`.

Converting a larger value type into a smaller value type is called *demotion,* whereas converting values in the opposite direction is known as *promotion.* Programmers say that the value of `int` variable `nVariable1` is promoted to a `double` in expressions such as the following:

```
int nVariable1 = 1;
double dVariable = nVariable1;
```

Naming conventions

You may have noticed that the name of each of the variables that I create begins with a special character that seems to have nothing to do with the name. These special characters are not special to C++ at all; they are merely meant to jog the reader's memory and indicate the type of the variable. A partial list of these special characters follows. Using this convention, I can immediately recognize `dVariable` as a variable of type `double`, for example.

Character	*Type*
`n`	`int`
`l`	`long`
`f`	`float`
`d`	`double`
`c`	`character`
`sz`	`string`

Religious wars worse than the "Great Palin Debate of 2008" have broken out over whether or not this naming convention clarifies C++ code. It helps me, so I stick with it. Try it for awhile. If after a few months, you don't think it helps, feel free to change your naming convention.

TIP

Mixed-mode expressions are not a good idea. Avoid forcing C++ to do your conversions for you.

Automatic Declarations

C++ '09 NEW

This entire section works only for C++ '09.

If you are really lazy, you can let C++ determine the types of your variables for you. Consider the following declaration:

```
int nVar = 1;
```

You might ask, "Why can't C++ figure out the type of `nVar`?" The answer is, as of C++ '09, it will if you ask nicely, as follows:

```
auto var1 = 1;
auto var2 = 2.0;
```

This says, “declare `var1` to be a variable of the same type as the constant value 1 (which happens to be an `int`) and declare `var2` to be the same type as 2.0 (which is a `double`).”

I consider the term `auto` to be a particularly unfortunate choice for this purpose because prior to C++ ’09, the keyword `auto` had a completely different meaning. However, `auto` had fallen out of use for at least ten years, so the standards people figured that it would be safe to usurp the term. Just be aware that if you see the keyword `auto` in some old code, you will need to remove it.

You can also tell C++ that you want a variable to be declared to be of the same type as another variable, whatever that might be, using the keyword `decltyple()`.

```
int var1;
decltype(var1) var2; // declare var2 to be of the
                     // same type as var1
```

C++ replaces the `decltype(var1)` with the type of `var1`, again an `int`.

Contents at a Glance

Table of Contents