IN THIS CHAPTER

- Finding out about C++
- Installing the Code::Blocks environment
- Building your first C++ program
- Executing the program
- Exploring a C++ program
- Creating an expression

Chapter 1
Writing Your First C++ Program

Okay, here we are: No one here but 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. 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 an Adobe Acrobat file ends in `pdf` 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 conversion, called *compiling,* is the job of the compiler. The machine code you wrote must be combined with some setup and teardown instructions and some standard library routines in a process known as *linking.* Together, compiling and linking are known as *building*. The resulting *machine-executable* files carry the extension `exe` in Windows. (They don't carry any particular extension in Linux or Macintosh, although a Macintosh might give the files the `app` extension.)

That sounds easy enough — 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 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 build the executable file.

Fortunately, there are public-domain C++ environments. I use one of them in this book — the Code::Blocks environment. This editor works with lots of different compilers, but the version of Code::Blocks combined with the gcc compiler used when writing this book is available for download for Windows. You can also find versions for various versions of Linux and an older version for the Macintosh.

The gcc compiler is maintained by the GNU Project. The compiler is part of a bigger suite of tools that work with a variety of programming languages including python, C, C++, Fortran, Ada, Go, and many more. You can find more on the GNU Project on the official GNU website at `https://www.gnu.org`. You can find more on the GNU Compiler Collection (GCC) and the gcc compiler at `https://gcc.gnu.org`. I cover more on the gcc compiler later in this chapter.

REMEMBER

Although Code::Blocks is public domain and listed at no cost, you're encouraged to pay a 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 Code::Blocks website (`www.codeblocks.org`) for details.

I have tested the programs in this book with Code::Blocks 25.03, which comes bundled with gcc version 14.2.0. This version of gcc implements most of the C++2026 standard.

C++26 NEW

You can use different versions of gcc or even different compilers, if you prefer, but they may not implement the complete C++ 26 standard. For that reason, 2026 extensions are marked with the C++ 26 icon shown in the margin.

Okay, I admit it: This book is somewhat Windows-centric. I have tested all the programs in the book on Windows 11. Many have also been tested on Ubuntu Linux as well as on Macintosh OS X. C++ is a cross-platform language, so regardless of which platform you use, if your compiler supports C++26, the code from this book should work.

WARNING

The Code::Blocks/gcc package generates 64-bit programs, but it doesn't easily support creating windowed programs. The programs in this book run from a command-line prompt and write out to the command line. As boring as that may sound, I strongly recommend that you work through the examples in this book first to learn C++ *before* you tackle windowed development. C++ and windowed programming are two separate concepts, 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 particular program's task is to convert a temperature value entered by the user from degrees Celsius to degrees Fahrenheit. (If you're using a different operating system such as macOS or Linux, tips for installing an IDE and C++ are also provided to help get you started.)

Installing Code::Blocks

You can download the most recent version of Code::Blocks from its website at `www.codeblocks.org/downloads/binaries`. There you can find the most recent version of the Code::Blocks environment for Windows, Ubuntu Linux, and Mac OS X. The following section describes installing Code::Blocks on Windows.

Microsoft Windows

The Code::Blocks environment comes in an easy-to-install, compressed executable file that's compatible with all versions of Microsoft Windows after Windows XP. (Though the site might not list Windows 11, it's supported.) Here's the rundown on installing the environment:

1. **Download the executable `codeblocks-25.03.mingw-setup.exe` from `www.codeblocks.org/downloads/binaries` and save the file to your desktop or another place where you can easily find it.**


TIP

Note that 25.03 might be a different number. As long as it's larger than 25.03, you're good to go!

This setup file includes a version of the gcc compiler.

2. **Double-click the program after it has completed downloading.**

This step starts the installation process.

3. **If (depending on which version of Windows you're using) you get the ubiquitous pop-up warning "An unidentified program wants access to your computer," click Allow to get the installation ball rolling.**

4. **Click Next after closing all extraneous applications, as you're warned in the Welcome dialog box to the Code::Blocks Setup Wizard.**

5. **Read the end user license agreement (EULA) and then click I Agree if you can live with its provisions.**

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

6. **Click the Next button.**

The installation program allows you to install only a subset of the features. You must select at least Default Install and the MinGW Compiler Suite. The default is to install everything — that's the best choice.


WARNING

If the MinGW Compiler Suite isn't an option, you must have downloaded a version of Code::Blocks that doesn't include gcc. This version won't work correctly.

7. **Click Install and accept the default destination folder.**

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

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

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

9. **From within Code::Blocks, choose Settings ➪ Compiler.**

   Doing so opens the Global Compiler Settings dialog box.

10. **Select the Compiler Flags tab.**

    A *compiler flag* is a special option that gets passed to the compiler to enable or disable features such as warnings, language standards, or debugging information. When you select the tab you'll see a long list of flag options with selection boxes on the right, as shown in Figure 1-1.

11. **Scroll to better see the options and make sure the following two flags are selected:**

    - Enable all common compiler warnings.
    - Have g++ follow the C++26 ISO C++ language standard.

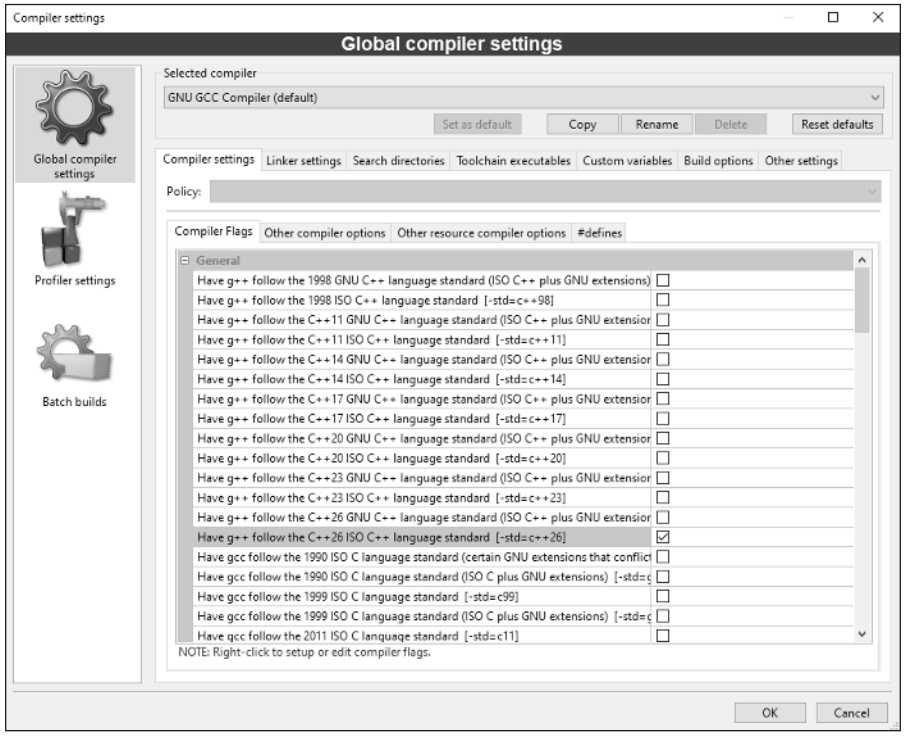

FIGURE 1-1: Ensure that the Enable All Compiler Warnings and the C++ 2026 flags are also set.

12. **In the same Global Compiler Settings dialog box, select the Toolchain Executables tab and make sure it looks like Figure 1-2.**

    The default location for the gcc compiler is the `MinGW\bin` folder within the `Code::Blocks` folder.

WARNING

If the default location is empty, Code::Blocks doesn't know where the gcc compiler is and cannot build your programs. Make sure you've downloaded a version of Code::Blocks that includes gcc and specified MinGW during the installation. If you're using an existing gcc compiler that you've already installed, you need to point Code::Blocks to wherever it's located on your hard drive.

13. **Close the Global Compiler Settings dialog box.**

14. **Click Next in the Code::Blocks Setup dialog box and then click Finish to complete the setup program.**

    The setup program takes the hint and exits.

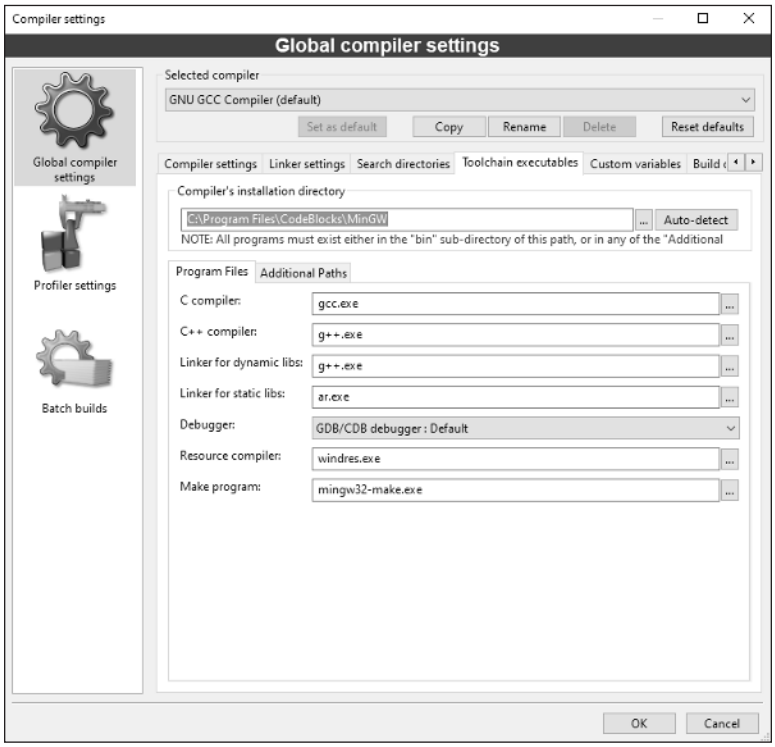

FIGURE 1-2: Ensure that the compiler's installation directory is correct.

Ubuntu Linux

Code::Blocks doesn't include gcc on Linux, so installation is a two-step process. First, you need to install gcc. Then you can install Code::Blocks.

WARNING

Depending on your system, when you install gcc or Code::Blocks, an older version might be installed. You need gcc 14.0 or later and Code::Blocks 25.03 or later in order to compile C++26 code.

The following section lists the default steps for installing these tools. If older versions are installed, you need to update the installation to the newer versions manually.

Installing gcc

The gcc compiler is readily available for Linux. Follow these steps to install it:

1. **Enter the following commands from a command prompt:**

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install g++
```

The standard Ubuntu Linux distribution includes a GNU C compiler but not the C++ extensions and, in particular, not the C++26 standard extensions. The first two commands update and upgrade the tools you already have. The third command installs C++.

2. **Enter the following command from a command prompt:**

```
gcc --version
```

If your GNU C++ version is 14.0 or later, you're set. If you have an earlier version, some of the C++ 2026 features may not work properly.

TIP

If you're using Debian Linux, the commands are the same. If you're using Red Hat Linux, replace the command `apt-get` with `yum` so that you end up with

```
sudo yum install g++
```

Installing Code::Blocks

If you cannot find Code::Blocks in the Software Center or another app store on your system, you should be able to follow these terminal steps to install Code::Blocks:

1. **If the Universe Repository option on your system isn't already enabled, enable it by entering the following lines:**

```
sudo add-apt-repository universe
sudo apt update
```

2. **Install Code::Blocks by entering the following command:**

```
sudo apt install codeblocks
```

   This step starts the installation process.

   Code::Blocks searches your hard drive for your C++ compiler. It should be able to find it without a problem, but if it doesn't, complete the following steps.

3. **Start Code::Blocks.**
4. **Choose Settings ➪ Compiler.**

   Doing so opens the Global Compiler Settings dialog box.

5. **Select the Compiler Flags tab.**
6. **Make sure the following two flags are selected:**
   - Enable all common compiler warnings
   - Have g++ follow the C++26 ISO C++ language standard
7. **In the same Global Compiler Settings dialog box, select the Toolchain Executables tab.**
8. **Click the Ellipsis (. . .) button at the right end of the field.**
9. **Navigate to `/usr`, unless you installed the gcc compiler somewhere other than the default location of `/user/bin`.**
10. **The C Compiler option should be gcc, the C++ Compiler option should be g++, and the Linker for Dynamic Libs option should be g++.**
11. **Click OK to close the dialog box.**

REMEMBER

When you start Code::Blocks, you should see version 25.03 or later or a date of 2025-03-29 or later. If an earlier version is shown, you need to consult either the Code::Blocks support site or your operating system's support site for information on updating your system to the latest version.

Macintosh

At the time this book was being written, Code::Blocks didn't have a Macintosh version that supported C++26. Fear not, however: It turns out that C++ is supported on macOS via Xcode, a free development package offered by Apple. To run the Xcode distribution from Apple for its compiler, you need to be running macOS Sequoia 15.6 or later.

Installing Xcode

Look for Xcode 16.3 or later in the Mac App Store or on the Apple Developer site at `https://developer.apple.com`. Note that the download and installation take quite some time because Xcode is several gigabytes in size.

After you've installed Xcode, you need to take one more step to ensure that all the C++ tools you need are installed. This extra step installs the command-line tools. To do this, open a terminal window and run the following command:

```
xcode-select --install
```

This should install all the standalone tools you need as well as some of the C++ support files. To confirm your C++ version, enter the following in the terminal:

```
clang++ --version
```

This command displays the version of C++ you're running. For C++26, you need to be running Xcode 16.3 or later.

Installing a development tool to use C++

Because the Macintosh version of Code::Blocks didn't support C++26 at the time of this writing, you need to use a different tool on your Mac. Suggested tools are JetBrains CLion (found at `www.jetbrains.com/clion`) or Microsoft's Visual Studio Code (found at `https://code.visualstudio.com/download`).

Creating Your First C++ Program

With all the necessary tools installed, it's time to create your first C++ program! That program will be all about converting Celsius temperatures into Fahrenheit, so I'll have you enter your C++ code into a file appropriately called `Conversion.cpp` and then have you convert the C++ code into an executable program.

Creating a project

The first step in creating any C++ program is to create what's 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 this book consist of a single source file and use a command-line style:

1. **Start up the Code::Blocks tool.**
2. **From within Code::Blocks, choose File ➪ New ➪ Project. . . from the main menu.**
3. **Select the Console Application icon and then click the Go button on the right side of the dialog.**

   The first time you carry out Step 3, you might be presented with a dialog box that welcomes you to the Console Application Wizard. Code::Blocks is simply being friendly! You can just click the Next button. Before clicking Next, you have the option to select the check box to prevent the dialog box from showing up again.

4. **From the next dialog box that appears, select C++ as the language you want to use and then click Next.**


REMEMBER

   Code::Blocks and gcc also support plain ol' C programs.

5. **In the Folder to Create Project In field, select the Ellipsis (. . .) button.**

   This step takes you to the file system dialog box, where you can specify where you want to create and save the project.

6. **Select Drive C on Windows — Windows (C:).**

   On Linux and Macintosh, you can select the desktop.

7. **Click the New Folder button.**

   It's likely near the top of the dialog box being displayed.

8. **Name the new folder** CPP_Programs_from_Book**.**

   The result should look like Figure 1-3. This is the folder where you save your programs. You can simply select it for any future programs you create from this book.

9. **Click the Select Folder button to continue.**

   This step returns you to the Console Application Wizard.

10. **In the Project Title field, type the name of the project.**

    In this case, type **Conversion**.

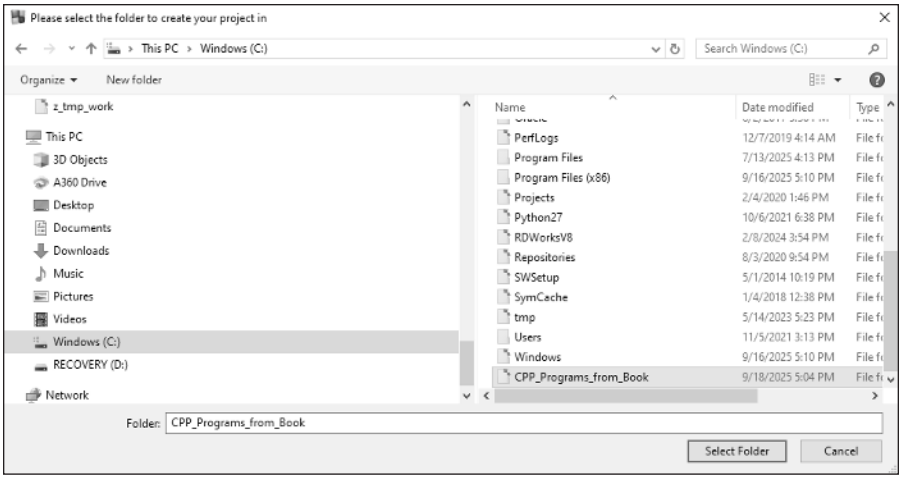

FIGURE 1-3: Put your project in the C:\CPP_Programs_from_Book folder on Windows.

The resulting screen is shown in Figure 1-4 on Windows. (The Linux and Macintosh versions should look the same, except for the path.)

11. Click the Next button.

The next dialog box gives you the option of creating an application for testing or the final version. The default values shown on this dialog are fine.

12. Click Finish to create the Conversion project.

FIGURE 1-4: Creating the Conversion project for your first program.

Entering the C++ code

The Conversion project that Code::Blocks initially creates consists of a single, default `main.cpp` file that displays the message `Hello, world`. The next step is to enter the program:

1. **In the Management dialog box on the left, double-click `main.cpp`, 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 1-5.

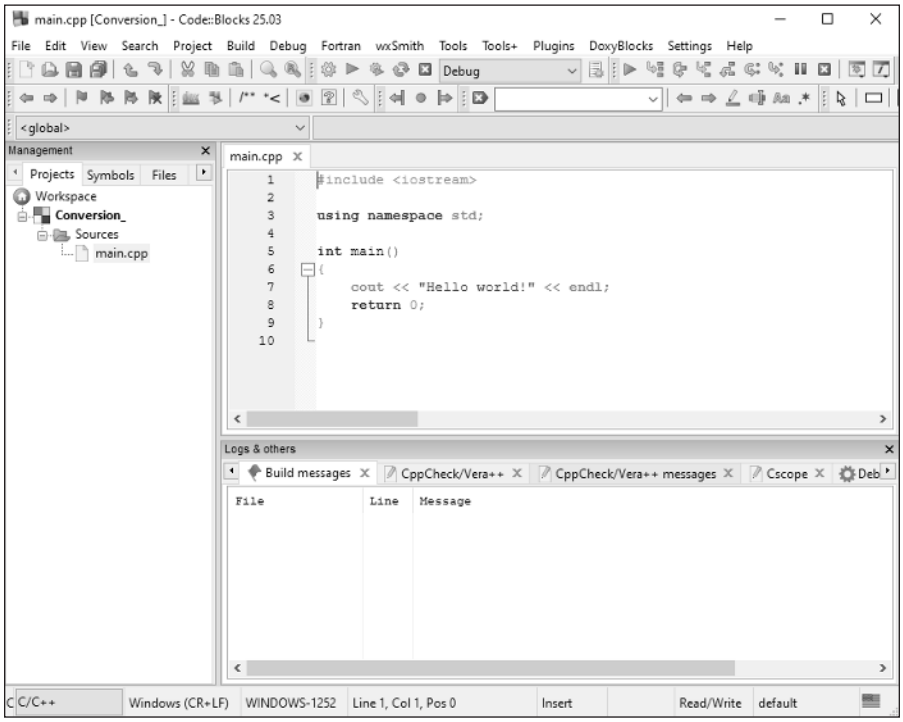

FIGURE 1-5: The Management dialog box displays a directory structure for all available programs.

2. **Edit `main.cpp` by replacing the code that is there with the code in this step 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 make sure everything is lowercase.

TIP

You can cheat by using the files at www.dummies.com/go/cplusplus8e, as described in the next section:

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

int main(int nNumberofArgs, char* pszArgs[])
{
   // enter the temperature in Celsius
   int celsius;
   print("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)
   print("Fahrenheit value is: ");
      println("{}", fahrenheit);

   return 0;
}
```

TECHNICAL STUFF

The lines in the code include comments to help you understand what the code is doing. These are the lines that start with double forward slashes (//). If you are typing the code in, you could technically skip including those lines and the program will still work just fine. Comments, however, make it clearer what the code is doing, so I strongly suggest you enter them as well.

3. **Choose File ➪ Save file from the main menu 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

You will find a downloadable file containing the programs in the book at `www.dummies.com/go/cplusplus8e`. Once you've downloaded the file, open it and do one of two things with the CPP_Programs_from_Book folder:

- Drag the files in it to the same CPP_Programs_from_Book folder that you created a few pages back in the "Creating a project" section; or
- If you haven't been through that section, copy the CPP_Programs_from_Book folder to the base folder of your computer's C: drive. All of the sources used in the book will then be found at `C:\CPP_Programs_from_Book`.

WARNING

You can put the CPP_Programs_from_Book folder at another location, but don't put your source files in a directory that includes a space in its name. On Windows, that means don't put any of your Code::Blocks folders in My Documents or on the desktop, because they both include a space in their paths.

You can use these files in two ways: One way is to complete the steps I describe in this book to create the program by hand first but then copy and paste from the provided files into your program if you run into trouble (or your fingers start cramping). This is the preferred technique.

A second approach is to use the sources provided and simply copy them into a new project each time:

1. **Double-click `AllPrograms.workspace` in `C:\CPP_Programs_from_Book`.**

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

2. **Right-click the Conversion project in the Management dialog box on the left, and then choose Activate Project from the menu that appears.**

   Code::Blocks turns the Conversion label bold to verify that this is the program you're working on right now. When you subsequently select Build, Code::Blocks always builds the active project.

3. **Double-click the `main.cpp` file under the Conversion project to open the file in the editor.**

As you navigate through his book and work on other listings, you'll often need to repeat Step 2, but each time you should select the appropriate project. The problem with this approach is that you tend to learn little about C++ if you don't enter the code yourself.

Building your program

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

To build your `Conversion` program, you choose Build ➪ Build from the main 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 near the bottom of the Logs & Others section, as shown in Figure 1-6.

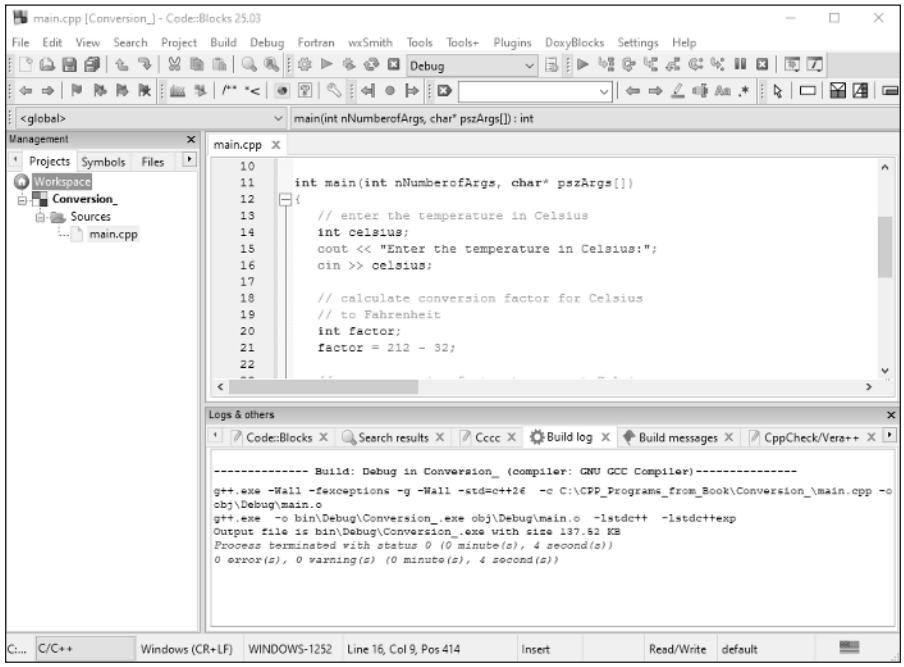

FIGURE 1-6: Code::Blocks builds the Conversion program quickly.

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, try introducing an error into your program — more specifically, change Line 16 from `cin >> celsius;` to `cin >>> celsius;`. You might think such a small change wouldn't be a big deal. It seems an innocent enough offense — forgivable to you and me perhaps, but not to C++. When you go on to choose Build ➪ Build to start the compile-and-build process. Code::Blocks almost immediately places a red square next to the erroneous line.

The message on the Build Message tab is a rather cryptic `error: expected primary-expression before ' >' token`. To get rid of the message, remove that extra `>` before `celsius` and recompile.

TIP

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

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 whether 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 `Conversion` program file and give it input to see how well it works.

To execute the `Conversion` program on Windows Code::Blocks, choose Build ➪ Build and Run from the main menu or press F9. This rebuilds the program if anything has changed and executes the program if the build is successful.

A dialog box 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:

```
Enter the temperature in Celsius:100
Fahrenheit value is:212
```

If you're using Code::Blocks, you also see a message to press any key to continue. The message gives you the opportunity to read what you've entered before it goes away. Press Enter (or any other key), and the dialog box (along with its contents) disappears. Congratulations! You just entered, built, and executed your first C++ program.

REMEMBER

Code::Blocks isn't truly intended for developing windowed programs like those used in Windows. 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.)

Windows programs show the user visually oriented output, all nicely arranged in onscreen windows. The 64-bit program `Conversion.exe` executes *under* Windows, but it's not a Windows program in the visual sense.

If you don't know what *64-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 Command Prompt box.

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

If you didn't see the Press Any Key message when you ran the `Conversion` program, but rather saw the output flash and then disappear, you need to add a few lines of code to pause the program at the end. Hold tight and I'll show you the code to add later in this chapter!

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 yourself. Programs are a bit like cars as well. All cars are basically the same with small differences and additions — okay, French cars are much different from 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 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 — one where, more often than not, you start with one or more statements that say *Include* followed by a main section. The code looks much 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 <iostream>
#include <print>
using namespace std;

int main(int nNumberofArgs, char* argv[])
{
   // Your C++ code starts here.

   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 with `main()`.

REMEMBER

I've copied this code into a file called `Template.cpp` located in the main `CPP_Programs_from_Book` folder.

If you're using a tool such as Code::Blocks, when you execute code using this structure, Code::Blocks follows the output with a prompt that tells you to press any key to continue. If it didn't do this, when your code runs, a window is opened, the output is displayed, the program ends, and the window then closes. This would happen so fast that you likely wouldn't see the output! Tools like Code::Blocks or Visual Studio Code step in to take care of you by stopping the window from closing.

If you're running code from a tool that doesn't do this or running your compiled program from a tool such as Windows Explorer that closes the output window immediately, you can add a few lines of code to each of your listings to stop this from happening. I've added such code (the bolded stuff) to the following template:

```
// TemplatePause - Provides a template to be used
//            as the starting point. Includes a prompt
//            to pause after running the program.
//
// The following include files define the majority of
// functions that any given program will need.
#include <iostream>
#include <print>
using namespace std;
```

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

   // Wait until user is ready before ending program
   // to allow the user to see the program results.
   println("Press Enter to continue...");
   cin.ignore(10, '\n');
   cin.get();

   return 0;
}
```

This code waits for the user to press a key before terminating the program. It displays a simple message, clears any previous key presses, and then waits for you to press any key before continuing. You can add these lines to the end of the main section of any listings within this book to pause the program.

I've copied this code as well into a file called `TemplatePause.cpp` located in the main `CPP_ Programs_from_Book` folder.

TIP

If the tools you use don't pause so that you can see the output, you should adjust the listings in this book to include the same three lines of code I had you add earlier to the `TemplatePause.cpp` file. As was the case with `TemplatePause.cpp`, these three lines are placed immediately in front of the `return` statement. If a particular listing doesn't have a `return` statement, the code is placed just before the closing bracket (`}`).

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 much smarter than I give it credit for. These first few lines, along with the first six lines in the `Template.cpp` program listing, are known as *comments:* These are the programmer's explanation of what they're 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* character, which is the character that terminates a command line. You can put any character you want in a comment. A comment can be as long as you want, but it's customary to limit comment lines to no more than 80 characters. Back in the old days ("old" is relative here), the code length for computer programs was limited to 80 characters. These days, limiting a single line to fewer than 80 characters is just a good practical idea (easier to read and 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.*

WARNING

C++ allows a second form of comment in which everything appearing after the /* characters and before the */ characters is ignored; however, this form of comment is normally no longer used in C++.

REMEMBER

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 for the programmer to explain their thinking when they wrote the code. A programmer's thoughts may not be obvious to the next colleague who tries to use or modify the program. In fact, the programmer may forget the program's purpose if they look at it months after writing the original code and the program 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 11 when you learn about the preprocessor.) Program execution begins with the first C++ statement after the open brace that appears after the line containing `main()`. Execution then continues through the listing, one statement at a time.

As you look through the program, you can see that spaces, tabs, and newlines appear in the program. These characters are collectively known as *whitespace* because you can't see them on the monitor.

TIP

You can 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 celsius;` is a declaration statement. A *declaration* is a statement that defines a variable. A *variable* is sort of 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 I 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 it can be used.

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.

TIP

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`.

TIP

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 `print`, `println`, `cout`, and `cin` are known as input/output statements, often contracted to *I/O statements.* (Like all engineers, programmers love contractions and acronyms.)

It's not hard to figure out the basic function of the `print` and `println` statements. Exactly — they print information to the standard output device. In this case, the

standard C++ output device is your monitor (also referred to as the *console* or "your screen"). The difference between `print` and `println` is that `println` adds a carriage return after printing so that the next time you print to the screen, the output starts on a new line.

I listed `cout`, but didn't actually use `cout` in the listing. I could have used the following code as an alternative way to print:

```
cout << "Enter the temperature in Celsius: ";
cin >> celsius;
  ...
cout << "Fahrenheit value is: ";
cout << Fahrenheit << endl;
```

C++23 NEW

The first I/O statement says, "Output the phrase *Enter the temperature in Celsius* to *cout*" (pronounced "see-out"), where *cout* is the name of the standard C++ output device. This does roughly the same thing as the print statements, but in a manner that's older and less safe. Starting with C++23, `print` and `println` were introduced as better ways to display output. As such, they're used throughout this book. (In Chapter 4, I dig deeper into explaining the printing functions and how to do more with them.)

The line that begins with `cin` is exactly the opposite of `cout` and of the printing lines. 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 you 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 piece of code that performs a calculation. Stated another way, an expression is a piece of code that can be evaluated to a value. An *operator* is a symbol that tells you how to combine or compare values within an expression to generate a value.

For example, in the `Conversion` program example — 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 $212-32$.

TIP

Unless you're taking a coding class, you likely will never have to explain the difference between a statement, an expression, and an operator. If you do ever need to explain the difference, a simple analogy is to think of a statement as a sentence, an expression as a phrase within that sentence, and an operator as a symbol within the expression.

Storing the results of an expression

The spoken language can be quite ambiguous. The term *equals* is one of those ambiguities. The word *equals* can indicate that two items have the same value, as in "A dollar equals 100 cents." The term *equals* can also imply assignment, as in math when you say that "*y* equals 3 times *x*."

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

```
fahrenheit = factor * celsius/100 + 32;
```

This expression uses the same mathematical symbols: the asterisk (`*`) for multiplication, the slash mark (`/`) for division, and the plus sign (`+`) 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 the expression 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 next 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.

The next three statements prompt the user to press Enter, and then they wait until Enter has been pressed. This is because on some systems the program can display the results and then close the console dialog box so rapidly that you don't even see that anything has happened.

TIP

As mentioned previously, on many systems, you can skip these three lines — Code::Blocks keeps the dialog box open until you press Enter anyway — but they don't hurt anything.

The final statement is `return 0`. This statement simply returns a value of `0`, as well as control, to your operating system. The `0` indicates that the program ran successfully and all is good in the world as far as this application is concerned.

And Now, a Few Comments on C++26

C++26 NEW

You can see that the main program presented in this book uses the `print` statements, which are from C++23 and refined in C++26. The book you're holding in your hands is about the current version of C++: C++26. As a major programming language, C++ has evolved over the years, and with recent updates, has become an even more modern language.

Because the focus of this book is on C++26, some older coding approaches are not used, although I'll talk about them throughout this book. This includes elements such as raw streams, `cout`, and raw pointers. C++ has gotten safer, but that means coding is a little different from what was done a decade ago. If you plan to use only an older C++ compiler or only the older stuff, the best solution is to buy the previous edition of this book. Again, because you'll see the older (less safe) ways of coding in existing C++ programs, I point it out throughout this book, just as I mention `cout` in this chapter.

If you were able to run the `Conversion` program with the print statements, you should be set with C++26.