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.

COPYRIGHTED MATERIAL
http://www.pbookshop.com

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.

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.

http://www.pbookshop.com

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

http://www.pbookshop.com

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.

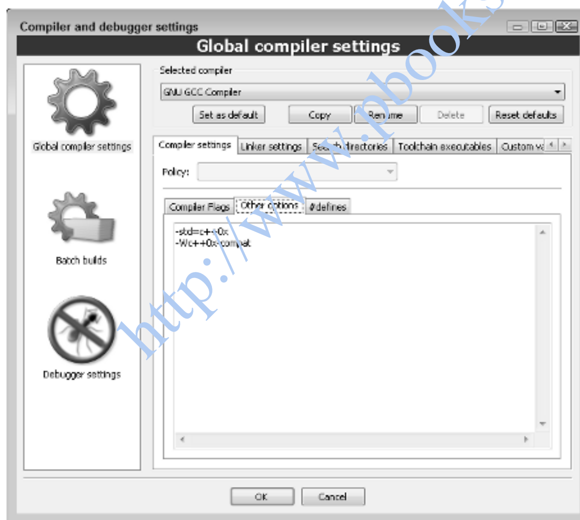
17. **Click OK.**

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

    The setup program exits.

http://www.pbookshop.com

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



http://www.pbookshop.com

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.

http://www.pbookshop.com

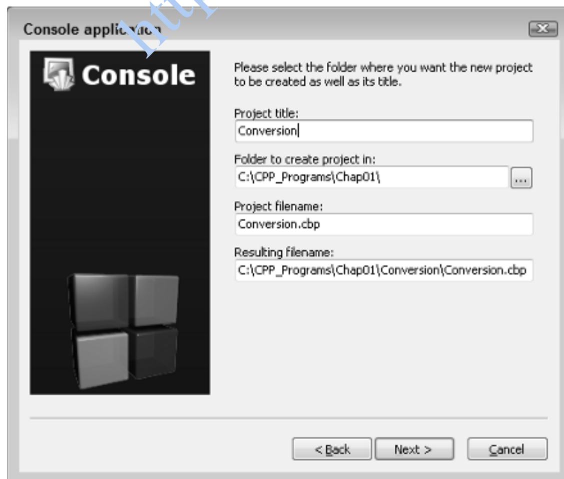


Figure 1-3: I created the project Conversion for the first program.

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.)

http://www.pbookshop.com

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.

http://www.pbookshop.com

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.

http://www.pbookshop.com

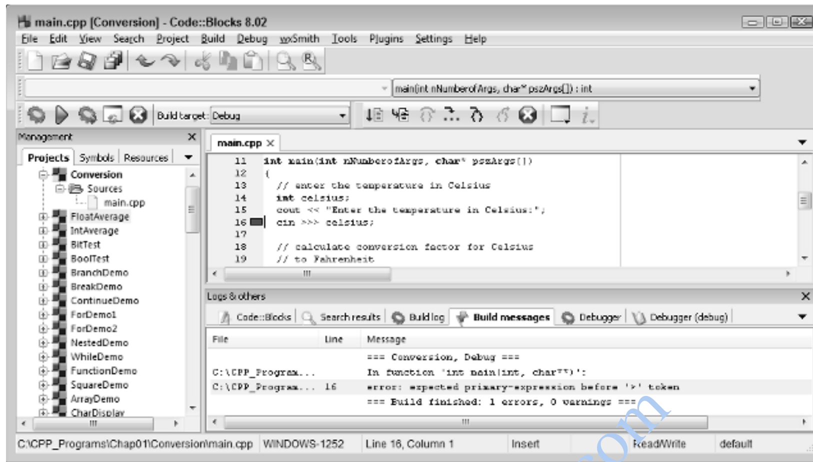
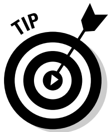


Figure 1-5: Code::Blocks flags the source of errors quickly.



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.

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.

http://www.pbookshop.com

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.

http://www.pbookshop.com

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.

http://www.pbookshop.com

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.

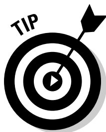
Basing programs on C++ statements

http://www.pbookshop.com

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
```

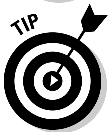
In the second expression, `y` is set equal to 3 times x, but what is `x`? The variable `x` acts as a holding tank for a value. In this case, the value of `x` is 10, but we could have just as well set the value of `x` to 20 or 30 or –1. The second formula makes sense no matter what the value of `x` is.

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.)

http://www.pbookshop.com

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`.

http://www.pbookshop.com

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."

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.

http://www.pbookshop.com