A Beginner's Guide That Makes You Feel **SMART**

C++ WITHOUT FEAR

THIRD EDITION

- **Learn** programming basics fast
- **Understand** with well-illustrated figures and examples
- **Practice** with games, exercises, and puzzles
- **Write** your first C++ program
- **Refer** to summaries, appendices, and C++14 notes

BRIAN OVERLAND

Updated for C++14!

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

C++ Without Fear

Third Edition

This page intentionally left blank

C++ Without Fear

Third Edition

A Beginner's Guide That Makes You Feel Smart

Brian Overland

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com

Library of Congress Cataloging-in-Publication Data

Overland, Brian, 1958– author.
C++ without fear : a beginner's guide that makes you feel smart / Brian Overland.—Third edition.
pages cm
Includes index.
ISBN 978-0-13-431430-3 (pbk. : alk. paper)—ISBN 0-13-431430-1 (pbk. : alk. paper)
1. C++ (Computer program language) I. Title.
QA76.73.C153O838 2016
005.13'3—dc23

2015033385

ISBN-13: 978-0-13-431430-3
ISBN-10: 0-13-431430-1
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

Once more, for Colin

This page intentionally left blank

Contents

This page intentionally left blank

Preface

It's safe to say that C++ is the most important programming language in the world today.

This language is widely used to create commercial applications, ranging from operating systems to word processors. There was a time when big applications had to be written in machine code because there was little room in a computer for anything else. But that time has long passed. Gone are the days in which Bill Gates had to squeeze all of BASICA into 64K!

C++, the successor to the original C language, remains true to the goal of producing efficient programs while maximizing programmer productivity. It typically produces executable files second in compactness only to machine code, but it enables you to get far more done. More often than not, C++ is the language of choice for professionals.

But it sometimes gets a reputation for not being the easiest to learn. That's the reason for this book.

We'll Have Fun, Fun, Fun…

Anything worth learning is worth a certain amount of effort. But that doesn't mean it can't be fun, which brings us to this book.

I've been programming in C since the 1980s and in C++ since the 1990s, and have used them to create business- and systems-level applications. The pitfalls are familiar to me—things like uninitialized pointers and using one equal sign (=) instead of two (==) in an "if" condition. I can steer you past the errors that caused me hours of debugging and sweat, years ago.

But I also love logic problems and games. Learning a programming language doesn't have to be dull. In this book, we'll explore the Tower of Hanoi and the Monty Hall paradox, among other puzzles.

Learning to program is a lot more fun and easy when you can visualize concepts. This book makes heavy use of diagrams and illustrations.

Why C and C++?

There's nothing wrong with other programming languages. I was one of the first people in the world to write a line of code in Visual Basic (while a project lead at Microsoft), and I admire Python as a high-level scripting tool.

But with a little care, you'll find C++ almost as easy to learn. Its syntax is slightly more elaborate than Visual Basic's or Python's, but C++ has long been seen as a clean, flexible, elegant language, which was why its predecessor, C, caught on with so many professionals.

From the beginning, C was designed to provide shortcuts for certain lines of code you'll write over and over; for example, you can use "++n" to add 1 to a variable rather than "n = n + 1." The more you program in C or C++, the more you'll appreciate its shortcuts, its brevity, and its flexibility.

C++: How to "Think Objects"

A systems programmer named Dennis Ritchie created C as a tool to write operating systems. (He won the Turing Award in 1983.) He needed a language that was concise and flexible, and could manipulate low-level things like physical addresses when needed. The result, C, quickly became popular for other uses as well.

Later, Bjarne Stroustrup created C++, originally as a kind of "C with classes." It added the ability to do object orientation, a subject I'll devote considerable space to, starting in Chapter 10. Object orientation is a way of building a program around intelligent data types. A major goal of this edition is to showcase object orientation as a superior, more modular way to program, and how to "think objects."

Ultimately, C++ became far more than just "C with classes." Over the years, support was added for many new features, notably the Standard Template Library (STL). The STL is not difficult to learn and this book shows you how to use it to simplify a lot of programming work. As time goes on, this library is becoming more central to the work of C++ programmers.

Purpose of the Third Edition

The purpose of the third edition is simple: double down on the strengths of past editions and correct limitations.

In particular, this edition aims at being more fun and easier to use than ever. Most of the features of the previous edition remain, but the focus is more on the practical (and entertaining) use of C++ and object orientation, and not as much on esoteric features that see little use. For example, I assume you won't want to write your own **string** class, because all up-to-date C++ compilers have provided this feature for a long time now.

In this edition, I also put more stress on "correct" programming practices that have become standard, or nearly so, in the C++ community.

This edition of the book starts out by focusing on successful installation and usage of the Microsoft C++ compiler, Community Edition. If you have another C++ compiler you're happy with, fine. You can use that because the great majority of examples are written in generic C++. The first chapter, however, guides you through the process of using the Microsoft compiler with Visual Studio, if you've never used them before.

Other features of this edition include:

- **Coverage of new features in C++11 and C++14:** This edition brings you up to date on many of the newest features introduced since C++11, as well as introducing some brand-new features in C++14. It's assumed you have a C++ compiler at least as up to date as the Microsoft Community Edition, so I've purged this edition of the book of some out-of-date programming practices.
- **Even more puzzles, games, exercises, and figures:** These features, all a successful part of the second edition, show up even more frequently in this edition.
- **More focus on the "whys" and "how tos" of object orientation:** The class and object features of C++ have always held great promise. A major goal in revising this edition was to put greater emphasis on the practical value of classes and objects, and how to "think objects."
- **More on the STL:** The Standard Template Library, far from being difficult to learn, can make your life much easier and make you more productive as a programmer. This edition explores more of the STL.
- **Useful reference:** This edition maintains the quick-reference appendixes in the back of the book and even expands on them.

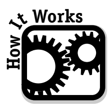
Where Do I Begin?

This edition assumes you know little or nothing about programming. If you can turn on a computer and use a menu system, keyboard, and mouse, you can begin with Chapter 1. I'll lead you through the process of installing and using Microsoft C++ Community version.

You should note that this version of C++ runs on recent versions of Microsoft Windows. If you use another system, such as a Macintosh, you'll need to download different tools. But the rules of generic C++ still apply, so you should be able to use most of the book without change.

Icons and More Icons

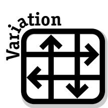
Building on the helpful icons in the first two editions, this edition provides even more—as signposts on the pages to help you find what you need. Be sure to look for these symbols because they call out sections to which you'll want to pay special attention.



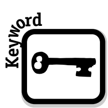
These sections take apart program examples and explain, line by line, how and why the examples work. You don't have to wade through long programming examples—I do that for you! (Or, rather, we go through the examples together.)

After each full programming example, I provide at least one exercise, and usually several, that build on the example in some way. These encourage you to alter and extend the programming code you've just seen. This is the best way to learn. The answers to the exercises can be found on my Web site (brianoverland.com).

These sections develop an example by showing how it can be improved, made shorter, or made more efficient.

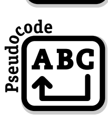


As with "Optimizing," these sections take the example in new directions, helping you learn by showing how the example can be varied or modified to do other things.



This icon indicates a place where a keyword of the language is introduced and its usage clearly defined. These places in the text summarize how a given keyword can be used.

The purpose of this icon is similar to "Keyword," but instead it calls attention to a piece of C++ syntax that does not involve a keyword.



"Pseudocode" is a program, or a piece of a program, in English-language form. By reading a pseudocode summary, you understand what a program needs to do. It then remains only to translate English-language statements into C++ statements.

This book also uses "Interludes," which are side topics that—while highly illuminating and entertaining—aren't always crucial to the flow of the discussion. They can be read later.

Note Finally, some important ideas are sometimes called out with notes; these notes draw your attention to special issues and occasional "gotchas." For example, one of the most common types of notes deals with version issues, pointing out that some features require a recent compiler:

C++14 This note is used to indicate sections that apply only to versions of C++ compliant with the more recent C++ specifications.

Anything Not Covered?

Nothing good in life is free—except maybe love, sunsets, breathing air, and puppies. (Well actually, puppies aren't free. Not long ago I looked at some Great Dane puppies costing around $3,000 each. But they were cute.)

To focus more on topics important to the beginner-to-intermediate programmer, this edition has slightly reduced coverage of some of the more esoteric subjects. For example, operator overloading (a feature you might never get around to actually programming into your classes) is still present but moved to the last chapter.

Most other topics—even relatively advanced topics such as bit manipulation—are at least touched upon. But the focus is on fundamentals.

C++ is perhaps the largest programming language on earth, much as English has the largest vocabulary of natural languages. It's a mistake for an introductory text to try to cover absolutely everything in a language of this size. But once you want to learn more about advanced topics in C++, there are plenty of resources.

Two of the books I'd recommend are Bjarne Stroustrup's *The C++ Programming Language*, *Fourth Edition* (Addison-Wesley, 2013), which is by the original author of the C++ language. This is a huge, sophisticated, and exhaustive text, and I recommend it after you've learned to be comfortable writing C++ code. As for an easy-to-use reference, I recommend my own *C++ for the Impatient* (Addison-Wesley, 2013), which covers nearly the whole language and almost every part of the Standard Template Library.

Graphical-user-interface (GUI) programming is specific to this or that platform and is deserving of its own—or rather many—books. This book introduces you to the core C++ language, plus its libraries and templates, which are platform independent.

A Final Note: Have Fun!

There's nothing to fear about C++. There are a few potholes here and there, but I'm going to steer you around them. Occasionally, C++ can be a little harder on you if you're not careful or don't know what you're doing, but you'll be better off in the long run by being made to think about these issues.

C++ doesn't have to be intimidating. I hope you use the practical examples and find the puzzles and games entertaining. This is a book about learning and about taking a road to new knowledge, but more than that, it's about enjoying the ride.

Acknowledgments

This edition is largely the result of a conversation between editor Kim Boedigheimer and myself while we had tea in a shop next to Seattle's Pike Place Market. So I think of this book as being as much hers as mine. She brought in an editorial and production team that made life easy for me, including Kesel Wilson, Deborah Thompson, Chris Zahn, Susan Brown Zahn, and John Fuller.

I'm especially indebted to Leor Zolman (yes, that's "Leor"), who provided the single finest technical review I've ever seen. Also providing useful input were John R. Bennett, a software developer emeritus from Microsoft, and online author David Jack ("the logic junkie"), who suggested some useful diagrams.

This page intentionally left blank

About the Author

Brian Overland published his first article in a professional math journal at age 14.

After graduating from Yale, he began working on large commercial projects in C and Basic, including an irrigation-control system used all over the world. He also tutored students in math, computer programming, and writing, as well as lecturing to classes at Microsoft and at the community-college level. On the side, he found an outlet for his lifelong love of writing by publishing film and drama reviews in local newspapers. His qualifications as an author of technical books are nearly unique because they involve so much real programming and teaching experience, as well as writing.

In his 10 years at Microsoft, he was a tester, author, programmer, and manager. As a technical writer, he became an expert on advanced utilities, such as the linker and assembler, and was the "go-to" guy for writing about new technology. His biggest achievement was probably organizing the entire documentation set for Visual Basic 1.0 and having a leading role in teaching the "object-based" way of programming that was so new at the time. He was also a member of the Visual C++ 1.0 team.

Since then, he has been involved with the formation of new start-up companies (sometimes as CEO). He is currently working on a novel.

This page intentionally left blank

1 Start Using C++

Nothing succeeds like success. This chapter focuses on successfully installing and using the C++ compiler—the tool that translates C++ statements into an executable program (or *application*).

I'm going to assume at first that you're using Microsoft Visual Studio, Community Edition. This includes an excellent C++ compiler—it's powerful, fast, and has nearly all of the up-to-date features. However, the Microsoft compiler raises some special issues, and one of the purposes of this chapter is to acquaint you with those issues so you can successfully use C++.

If you're not using this compiler, skip ahead to the section, "If You're Not Using Microsoft."

I'll get into the more abstract aspects of C++ later, but first let's get that compiler installed.

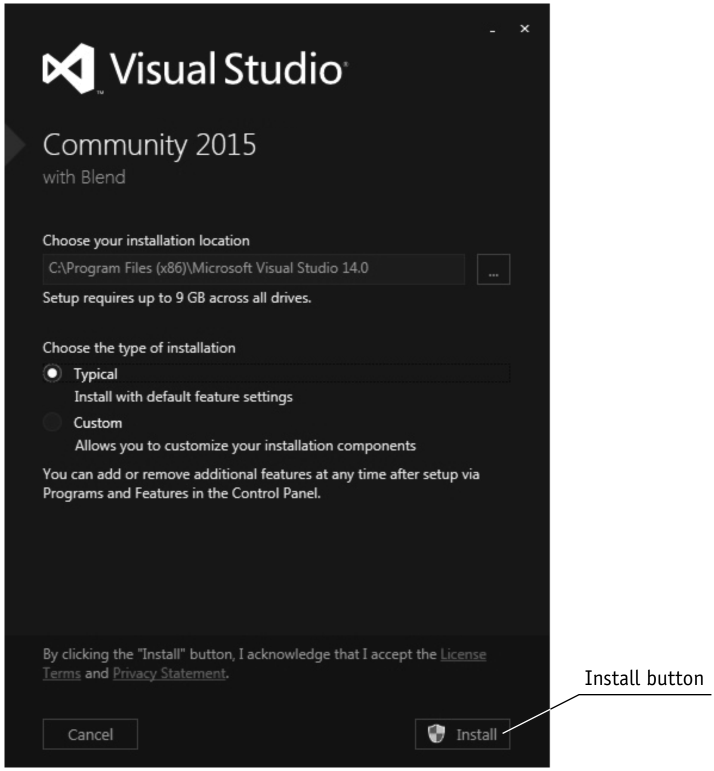
Install Microsoft Visual Studio

Even if you have an older version of Microsoft Visual Studio, you should consider updating to the current Community Edition, because it has nearly all the up-to-date features presented in this book. If you're already running Enterprise Edition, congratulations, but make sure it's up to date.

Here are the steps for installing Microsoft Visual Studio Community Edition:

1. Regardless of whether you're downloading from the Internet (you can use a search engine to look up "Visual Studio download") or, using the CD accompanying this book's Barnes & Noble Special Edition, get a copy of the file **vc_community** on your computer. If you're downloading, this will be found in your Download folder after using the site.

2. Double click the file **vc_community**. This launches the installation program. The following screen appears:



Used with permission from Microsoft.

3 Click the Install button in the lower-right corner. Installation should begin right away.

4 If you're downloading from the Internet, be prepared for a long wait! If you're using the CD, installation will be many, many times faster.

If all goes well, Microsoft Visual Studio, which includes the Microsoft C++ compiler, should be installed on your computer, and you're ready to start programming. First, however, you need to create a project.

Create a Project with Microsoft

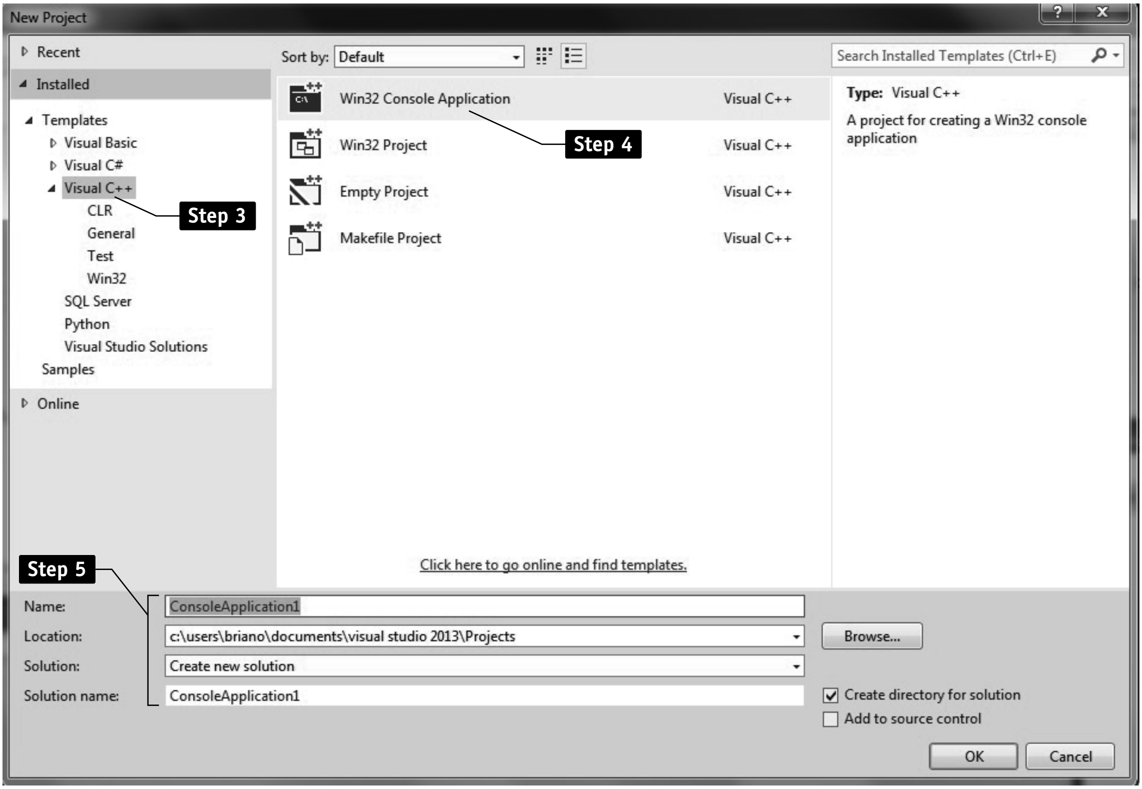
There are some files and settings you need for even the simplest program, but Visual Studio puts all the items you need into something called a *project*.

With Visual Studio, Microsoft makes things easy by providing everything you need when you create a project. Note that you will need to create a new project for each program you work on.

So let's create a project.

1 Launch Visual Studio. After you've installed it, you should find that Visual Studio is available on the Start menu (assuming you're running Windows). Visual Studio should then appear onscreen.

2 From the File menu (the first menu on the menu bar), choose the New Project command. The New Project window then appears.



Used with permission from Microsoft.

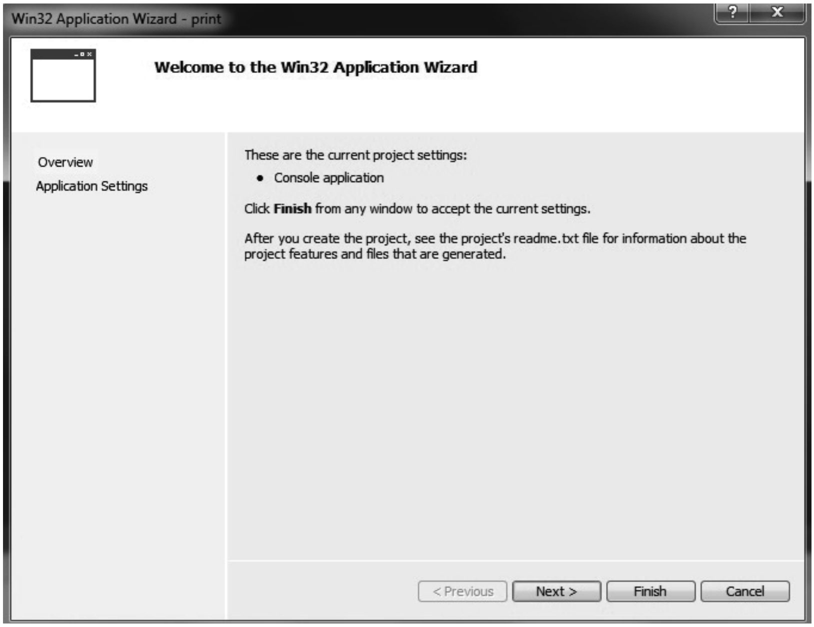
3 In the left pane, select Visual C++.

4 In the central windowpane, select Win32 Console Application.

5 There are four text boxes at the bottom of the window. You need only fill out one. In the Name box, type the name of the program: in this case, "print1." The Solution name box will automatically display the same text.

6 Click OK in the bottom right corner or just press ENTER.

The Application Wizard appears, asking if you're ready to go ahead. (Of course you are.) Click the Finish button at the bottom of the window.



Used with permission from Microsoft

After you complete these steps, a new project is opened for you. The major area on the screen is a text window into which you can enter a program. Visual Studio provides a skeleton (or boilerplate) for a new program containing the following:

```
// print1.cpp: Defines the entry point...
//

#include "stdafx.h"

int _tmain(int arg, _TCHAR* argv[])
{
     return 0;
}
```

You're probably asking, what is all this stuff? The first thing to be aware of is that any line that begins with double slashes (//) is a *comment* and is ignored by the compiler.

Comments exist for the benefit of the programmer, presumably to help a human read and understand the program better, but the C++ compiler completely ignores comments. For now, we're going to ignore them as well.

So the part you care about is just:

```
#include "stdafx.h"

int _tmain(int arg, _TCHAR* argv[])
{
     return 0;
}
```

Writing a Program in Microsoft Visual Studio

Now—again, assuming you're using Microsoft Visual Studio—you're ready to write your first program. The previous section showed the skeleton (or boilerplate) that's already provided. Your task is to insert some new statements.

In the following example, I've added the new lines and placed them in bold—so you know exactly what to type:

```
#include "stdafx.h"

#include <iostream>
using namespace std;

int _tmain(int arg, _TCHAR* argv[])
{
     cout << "Never fear, C++ is here!";
     return 0;
}
```

For now, just leave **#include "stdafx.h"** and **t_main** alone, but add new statements where I've indicated. These lines are Microsoft specific, and I'll have more to say about them in the section "Compatibility Issue #1: stdafx.h." First, however, let's just run the program.

Running a Program in Visual Studio

Now you need to translate and run the program. In Visual Studio, all you do is press Ctrl+F5 or else choose the Start Without Debugging command from the Debug menu.

Visual Studio will say that the program is out of date and ask if you want to rebuild it. Say yes by clicking the Yes button.

Note You can also build and run the program by pressing F5, but the output of the program will "flash" and not stay on the screen. So use Ctrl+F5 instead.

If you received error messages, you probably have mistyped something. One of the intimidating aspects of C++, until you get used to it, is that even a single mistyped character can result in a series of "cascading errors." So, don't get upset, just check your spelling. In particular, check the following:

- The two C++ statements (and most lines of code you type in *will* be C++ statements), end with a semicolon (;), so be careful not to forget those semis.
- But make sure the **#include** directives do *not* end with semicolons(;).
- Case sensitivity absolutely matters in C++ (although spacing, for the most part, does not). Make sure you did not type any capital letters except for text enclosed in quotation marks.

After you're sure you've typed everything correctly, you can rebuild the program by pressing Ctrl+F5 again.

Compatibility Issue #1: stdafx.h

If you're like me, you'd prefer not to deal with compatibility issues but get right to programming. However, there are a couple of things you need to keep in mind to make sure you succeed with Microsoft Visual Studio.

In order to support something called "precompiled headers," Microsoft Visual Studio inserts the following line at the beginning of your programs. There's nothing wrong with this, unless you paste sample code over it and then wonder why nothing works.

```
#include "stdafx.h"
```

The problem is that other compilers will not work with this line of code, but programs built with Microsoft Visual Studio require it, unless you make the changes described in this section.

You can adopt one of several strategies to make sure your programs compile inside Microsoft Visual Studio.

- The easiest thing to do is to make sure this line of code is always the first line in any program created with Visual Studio. So, if you copy generic C++ code listings into a Visual Studio project, make sure you do not erase the directive **#include "stdafx.h"**.
- If you want to compile generic C++ code (nothing Microsoft-specific), then, when creating a project, do not click the Finish button when the Application Wizard window appears. Instead, click Next. Then, in the Application Settings window, click the "Precompiled Headers" button to de-select it.
- After a project is created, you can still change settings by doing the following: First, from the Project menu, choose the Properties command (Alt + F7). Then, in the left pane, select Precompiled Headers. (You may first have to expand "Configuration Properties" and then expand "C/C++" by clicking on these words.) Finally, in the right pane, choose "Not Using Precompiled Headers" from the top drop-down list box.

With the last two options, Microsoft-specific lines such as **#include "stdafx.h"** still appear! However, after the Precompiled Headers option box is de-selected, the Microsoft-specific lines can be replaced with generic C++ code.

Also note that Visual Studio uses the following skeleton for the main function:

```
int _tmain(int arg, _TCHAR* argv[])
{

}
```

instead of:

```
int main()
{

}
```

Both of these work fine with Visual Studio, but if you keep the version that features the word **_tmain**, remember that it requires **#include stdafx.h** as well.

The items inside the parentheses, just after **_tmain**, support access to command-line arguments. But since this book does not address command-line arguments, you won't need them for the examples in this book. Just leave them as they are.

Compatibility Issue #2: Pausing the Screen

As stated earlier, if you build and run the program by pressing Ctrl+F5, your results should be satisfactory, but if you press F5, you'll get the problem of the program output flashing on the screen and disappearing.

If you're using Microsoft Visual Studio, the easiest solution is to just press Ctrl+F5 (Start Without Debugging) every time you build and run the program. However, not all compilers offer this option.

Another way to deal with the problem of output flashing on the screen and disappearing is to add the following line of code, just above "return 0;":

```
system("PAUSE");
```

When this statement is executed, it has roughly the same effect as pressing Ctrl+F5. It causes the program to pause and print "Press any key to continue."

The problem with this statement is that it is system specific. It does what you want in Windows, but it might not work on another platform. Only put this statement in if you're reasonably sure you want your program to run just on Windows-based systems.

If you're working on another platform, you'll need to look for another solution. Check your compiler documentation for more information.

Now, if you're using Microsoft Visual Studio, skip ahead to Exercise 1.1.

If You're Not Using Microsoft

If you're not using Microsoft Visual Studio as your compiler, most of the steps described in the previous sections won't apply. If any documentation comes with your compiler, make sure you read it in case, like Microsoft Visual Studio, it has idiosyncrasies of its own.

With compilers other than Visual Studio, do not put in the line **#include "stdafx.h"** and make sure you use the simpler program skeleton:

```
int main() {

}
```

Beginning with the next section, this book is going to adhere fairly closely to generic C++, which has nothing that is platform or vendor specific. But in this chapter, I'll keep reminding you of what you need to do for Visual Studio.

Example 1.1. *Print a Message*

Here is the program introduced earlier, written in generic C++ (except for the comment, which indicates what you have to do to run it in Visual Studio).

print1.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
     cout << "Never fear, C++ is here! ";
     return 0;
}
```

Remember that exact spacing does not matter, but case-sensitivity does.

Also remember that if and only if you are working with Microsoft Visual Studio, then, at the beginning of the program, you must leave in the following line:

```
#include "stdafx.h"
```

After entering the program, build and run it (from within Microsoft Visual Studio, press Ctrl+F5). Here's what the program prints when correctly entered and run:

```
Never fear, C++ is here!
```

However, this output may be run together with the message "Press any key to continue." In the upcoming sections, we're going to correct that.

How It Works

Believe it or not, this simple program has only one real statement. You can think of the rest as "boilerplate" for now—stuff you have to include but can safely ignore. (If you're interested in the details, the upcoming "Interlude" discusses the **#include** directive.)

Except for the one line in italics, the lines below are "boilerplate": these are items that always have to be present, even if the program doesn't do anything. For now, don't worry about why these lines are necessary; their usage will become clearer as you progress with this book. In between the braces ({}), you insert the actual lines of the program—which in this case consist of just one important statement.

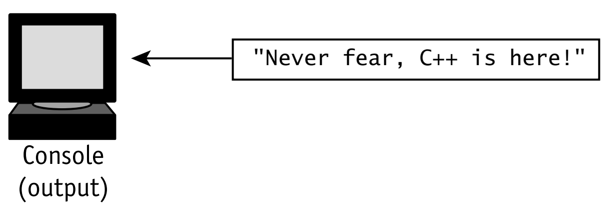
```
#include <iostream>
using namespace std;

int main()
{
     Enter_your_statements_here!
     return 0;
}
```

This program has one only real statement. Don't forget the semicolon (;) at the end!

```
cout << "Never fear, C++ is here!";
```

What is **cout**? This is an *object*—that's a concept I'll discuss a lot more in the second half of the book. In the meantime, all you have to know is that **cout** stands for "console output." In other words, it represents the computer screen. When you send something to the screen, it gets printed, just as you'd expect.

In C++, you print output by using **cout** and a leftward stream operator (<<) that shows the flow of data from a value (in this case, the text string "Never fear, C++ is here!") to the console. You can visualize it this way:




```
cout    <<    "Never fear, C++ is here! " ;
```

Don't forget the semicolon (;). Every C++ statement must end with a semicolon, with few exceptions.

For technical reasons, **cout** must always appear on the left side of the line of code whenever it's used. Data in this case flows to the left. Use the leftward "arrows," which are actually a pair of less-than signs (<<).

The following table shows other simple uses of **cout**:

STATEMENT	ACTION
`cout << "Do you C++?";`	Prints the words "Do you C++?"
`cout << "I think,";`	Prints the words "I think,"
`cout << "Therefore I program.";`	Prints the words "Therefore I program."

EXERCISES

Exercise 1.1.1. Write a program that prints the message "Get with the program!" If you want, you can work on the same source file used for the featured example and alter it as needed. (Hint: Alter only the text inside the quotation marks; otherwise, reuse all the same programming code.)

Exercise 1.1.2. Write a program that prints your own name.

Exercise 1.1.3. Write a program that prints "Do you C++?"

Interlude

What about the #include and using?

I said that the fifth line of the program is the first "real" statement of the program. I glossed over the first line:

```
#include <iostream>
```

This is an example of a C++ *preprocessor directive*, a general instruction to the C++ compiler. A directive of the form

```
#include <filename>
```

loads declarations and definitions that support part of the C++ standard library. Without this directive, you couldn't use **cout**.

If you've used older versions of C++ and C, you may wonder why no specific file (such as an .h file) is named. The filename iostream is a *virtual include file*, which has information in a precompiled form.

If you're new to C++, just remember you have to use **#include** to turn on support for specific parts of the C++ standard library. Later, when we start using math functions such as **sqrt** (square root), you'll need to switch on support for the math library:

```
#include <cmath>
```

▼ continued on next page

Interlude

▼ *continued*

Is this extra work? A little, yes. Include files originated because of a distinction between the C language and the standard runtime library. (Professional C/C++ programmers sometimes avoid the standard library and use their own.) Library functions and objects—although they are indispensable to beginners—are treated just like user-defined functions, which means (as you'll learn in Chapter 4) that they have to be declared. That's what include files do.

You also need to put in a **using** statement. This enables you to refer directly to objects such as **std::cout**. Without this statement, you'd have to print messages this way:

```
std::cout << "Never fear, C++ is here!";
```

We're going to be using **cout** (and its cousin, **cin**) quite a lot, so for now it's easier just to put a **using** statement at the beginning of every program.

Advancing to the Next Print Line

With C++, text sent to the screen does not automatically advance to the next physical line. You have to print a *newline* character to do that. (Exception: If you never print a newline, the text may automatically "wrap" when the current physical line fills up, but this produces an ugly result.)

The easiest way to print a newline is to use the predefined constant **endl**. For example:

```
cout << "Never fear, C++ is here!" << endl;
```

Note ▶ The endl name is short for "end line"; it is therefore spelled "end ELL," not "end ONE." Also note that **endl** is actually **std::endl**, but the **using** statement saves you from having to type **std::**.

Another way to print a newline is to insert the characters \n. This is an escape sequence, which C++ interprets as having a special meaning rather than interpreting it literally. The following statement has the same effect as the previous example:

```
cout << "Never fear, C++ is here!\n";
```

Example 1.2. *Print Multiple Lines*

The program in this section prints messages across several lines. If you're following along and entering the programs, remember once again to use uppercase and lowercase letters exactly as shown—although you can change the capitalization of the text inside quotation marks and the program will still run.

If you're working with Visual Studio, the only lines you should add are the ones shown here in bold. Leave **#include stdafx.h** and **_tmain** alone. If you're working with another compiler, the code should look as follows, minus the comments (//).

print2.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
     cout << "I am Blaxxon," << endl;
     cout << "the godlike computer." << endl;
     cout << "Fear me!" << endl;
     return 0;
}
```

Remember that exact spacing does not matter, but case-sensitivity does.

The resulting program, if you're working with Visual Studio, should be as follows. The lines in bold are what you need to add to the code Visual Studio provides for you.

```
#include "stdafx.h"

#include <iostream>
using namespace std;

int _tmain(int arg, _TCHAR* argv[])
{
     cout << "I am Blaxxon," << endl;
     cout << "the godlike computer." << endl;
```

```
        cout << "Fear me!" << endl;
        return 0;
    }
```

After entering the program, compile and run it. Here's what the program prints when correctly entered and run:

```
I am Blaxxon,
the godlike computer.
Fear me!
```

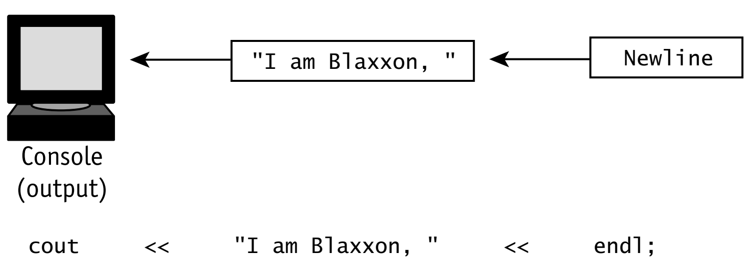
How It Works

This example is similar to the first one I introduced. The main difference is this example uses newline characters. If these characters were omitted, the program would print

```
I am Blaxxon, the godlike computer.Fear me!
```

which is not what we wanted.

Conceptually, here's how the statements in the program work:



You can print any number of separate items this way, though again, they won't advance to the next physical line without a newline character (**endl**). You could send several items to the console with one statement

```
cout << "This is a " << "nice " << "C++ program.";
```

which prints the following when run:

```
This is a nice C++ program.
```

Or, you can embed a newline, like this

```
cout << "This is a" << endl << "C++ program.";
```

which prints the following:

```
This is a
C++ program.
```

The example, like the previous one, returns a value. “Returning a value” is the process of sending back a signal—in this case to the operating system or development environment.

You return a value by using the **return** statement:

```
return 0;
```

The return value of **main** is a code sent to the operating system, in which 0 indicates success. The examples in this book return 0, but they could return an error code sometimes (–1 for example) if you found that to be useful. However, I would ignore that for now.

Exercises

EXERCISES

Exercise 1.2.1. Remove the newlines from the example in this section, but put in extra spaces so that none of the words are crammed together. (Hint: Remember that C++ doesn’t automatically insert a space between output strings.) The resulting output should look like this:

```
I am Blaxxon, the godlike computer. Fear me!
```

Exercise 1.2.2. Alter the example so that it prints a blank line between each two lines of output—in other words, make the results double-spaced rather than single-spaced. (Hint: Print *two* newline characters after each text string.)

Exercise 1.2.3. Alter the example so that it prints two blank lines between each of the lines of output.

Interlude

What Is a String?

From the beginning, I’ve made use of text inside of quotes, as in this statement:

```
cout << "Never fear, C++ is here!";
```

Everything outside of the quotes is part of C++ syntax. What’s inside the quotes is data.

▼ *continued on next page*

Interlude

▼ continued

In actuality, all the data stored on a computer is numeric, but depending on how data is used, it can be interpreted as a string of printable characters. That's the case here.

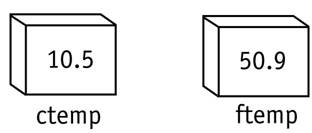
You may have heard of "ASCII code." That's what kind of data "Never fear, C++ is here!" is in this example. The characters "N", "e", "v", "e", "r", and so on, are stored in individual bytes, each of which is a numeric code corresponding to a printable character.

I'll talk a lot more about this kind of data in Chapter 8. The important thing to keep in mind is that text enclosed in quotes is considered raw data, as opposed to a command. This kind of data is considered a string of text or, more commonly, just a *string*.

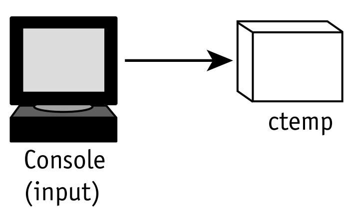
Storing Data: C++ Variables

If all you could do was print messages, C++ wouldn't be useful. The fundamental purpose of nearly any computer program is usually to get data from somewhere—such as end-user input—and then do something interesting with it.

Such operations require *variables*. These are locations into which you can place data. You can think of variables as magic boxes that hold values. As the program proceeds, it can read, write, or alter these values as needed. The upcoming example uses variables named ctemp and ftemp to hold Celsius and Fahrenheit values, respectively.



How are values put into variables? One way is through console input. In C++, you can input values by using the **cin** object, representing (appropriately enough) console input. With **cin**, you use a stream operator showing data flowing to the right (>>):




```
cin     >>     ctemp ;
```

Here's what happens in response to this statement. (The actual process is a little more complicated, but don't worry about that for now.)

1 The program suspends running and waits for the user to enter a number.

2 The user types a number and presses ENTER.

3 The number is accepted and placed in the variable ctemp (in this case).

4 The program resumes running.

So, if you think about it, a lot happens in response to this statement:

```
cin >> ctemp;
```

But before you can use a variable in C++, you must declare it. This is an absolute rule and it makes C++ different from Basic, which is sloppy in this regard and doesn't require declaration (but generations of Basic programmers have banged their heads against their terminals as they discovered errors cropping up as a result of Basic's laxness about variables).

This is important enough to justify restating, so I'll make it a cardinal rule:

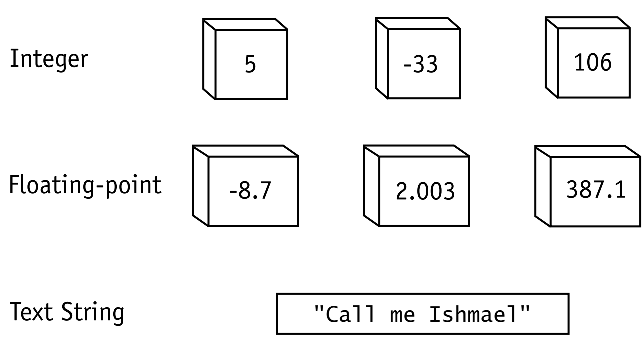
In C++, you must declare a variable before using it.

To declare a variable, you first have to know what *data type* to use. This is a critical concept in C++ as in most other languages.

Introduction to Data Types

A variable is something you can think of as a magic box into which you can place information—or, rather, *data*. But what kind of data?

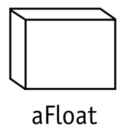
All data on a computer is ultimately numeric, but it is organized into one of three basic formats: integer, floating-point, and text string.



There are several differences between floating-point and integer format. But the main rule is simple:

If you need to store numbers with fractional portions, use a floating-point variable; otherwise, use integer types.

The principal floating-point data type in C++ is **double**. This may seem like a strange name, but it stands for "double-precision floating point." There is also a single-precision type (**float**), but its use is relatively infrequent. When you need the ability to retain fractional portions, you'll get better results—and fewer error messages—if you stick to **double**.



A **double** declaration has the following syntax. Note that this statement is terminated with a semicolon (;), just as most kinds of statements are.

Keyword

```
double variable_name;
```

You can also use a **double** declaration to create a series of variables:

```
double variable_name1, variable_name2, ...;
```

For example, this statement creates a **double** variable named aFloat:

```
double  aFloat;
```

This statement creates a variable of type **double**.

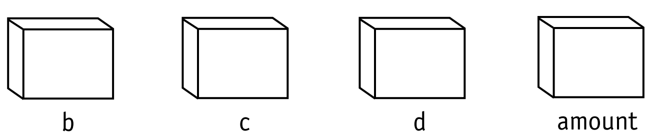
The next statement declares four **double** variables named b, c, d, and amount:

```
double  b, c, d, amount;
```

The effect of this statement is equivalent to the following:

```
double  b;
double  c;
double  d;
double  amount;
```

The result of these declarations is to create four variables of type **double**.



An important rule of good programming style is that variables should usually be initialized, which means giving them a value as soon as you declare them. The declarations just shown should really be:

```
double  b  = 0.0;
double  c  = 0.0;
double  d  = 0.0;
double  amount = 0.0;
```

Starting in the next chapter, I'll have a lot more to say about issues such as data types and initialization. But for the next program, I'll keep the code simple. We'll worry about initialization in Chapter 2 onward.

Interlude

Why Double Precision, Not Single?

Double precision is like single precision, except better. Double precision supports a greater range of values, with better accuracy: It uses 8 bytes rather than 4.

C++ converts all data to double precision when doing calculations, which makes sense given that today's PCs include 8-byte co-processors. C++ also stores floating-point constants in double precision unless you specify otherwise (for example, by using the notation 12.5F instead of 12.5).

Double precision has one drawback: it requires more space. This is a factor only when you have large amounts of floating-point values to be stored in a disk file. Then, and only then, should you consider using the single-precision type, **float**.

Example 1.3. *Convert Temperatures*

Every time I go to Canada, I have to convert Celsius temperatures to Fahrenheit in my head. If I had a handheld computer, it would be nice to tell it to do this conversion for me; computers are good at that sort of thing.

Here's the conversion formula. The asterisk (*), when used to combine two values, means "multiply by."

```
Fahrenheit = (Celsius * 1.8) + 32
```

Now, a useful program will take *any* value input for Celsius and then convert it. This requires the use of some new features:

- Getting user input
- Storing the value input in a variable

Here is the complete program. Create a new project called "convert." Then enter the new program, and compile and run (press Ctrl + F5 if you're using Microsoft).

convert.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
     double  ctemp, ftemp;

     cout << "Input a Celsius temp and press ENTER: ";
     cin >> ctemp;
     ftemp = (ctemp * 1.8) + 32;
     cout << "Fahrenheit temp is: " << ftemp;
     return 0;
}
```

Remember, yet again (!), that if and only if you're working with Microsoft Visual Studio, you must leave the following line in at the beginning of the program:

```
#include "stdafx.h"
```

Programs are easier to follow when you add comments, which in C++ are notated by double slashes (//). Comments are ignored by the compiler (they have no effect on program behavior), but they are useful for humans. Here is the more heavily commented version:

convert2.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;
```

convert2.cpp, cont.

```
int main()
{

    double ctemp;   // Celsius temperature
    double ftemp;   // Fahrenheit temperature

    // Get value of ctemp (Celsius temp).

    cout << "Input a Celsius temp and press ENTER: ";
    cin >> ctemp;


    // Calculate ftemp (Fahrenheit temp) and output.

    ftemp = (ctemp * 1.8) + 32;
    cout << "Fahrenheit temp is: " << ftemp << endl;

    return 0;
}
```

This commented version, although it's easier for humans to read, takes more work to enter. While following the examples in this book, you can always omit the comments or choose to add them later. Remember this cardinal rule for comments:

C++ code beginning with double slashes (//) is a comment and is ignored by the C++ compiler to the end of the line.

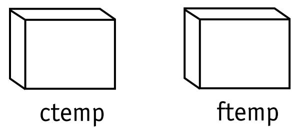
Using comments is always optional, although it is a good idea, especially if any humans (including you) are going to ever look at the C++ code.

How It Works

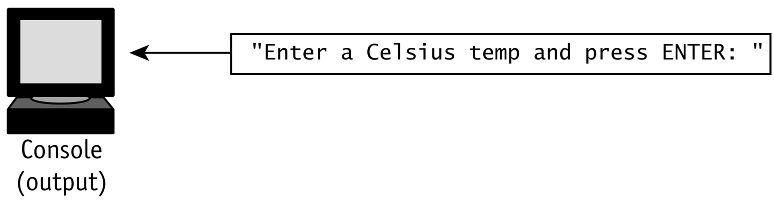
The first statement inside **main** declares variables of type **double**, ctemp and ftemp, which store Celsius temperature and Fahrenheit temperature, respectively.

```
double  ctemp, ftemp;
```

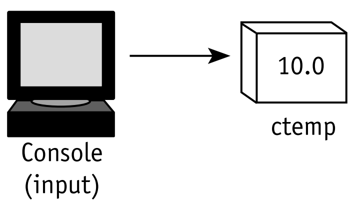
This gives us two locations at which we can store numbers. Because they have type **double**, they can contain fractional portions. Remember that **double** stands for "double-precision floating point."



The next two statements prompt the user and then store input in the variable ctemp. Assume that the user types 10. Then the numeric value 10.0 is put into ctemp.




```
cout    <<    "Enter a Celsius temp and press ENTER: " ;
```




```
cin     >>     ctemp;
```

In general, you can use similar statements in your own programs to print a prompting message and then store the input. The prompt is very helpful because otherwise the user may not know when he or she is supposed to do something.

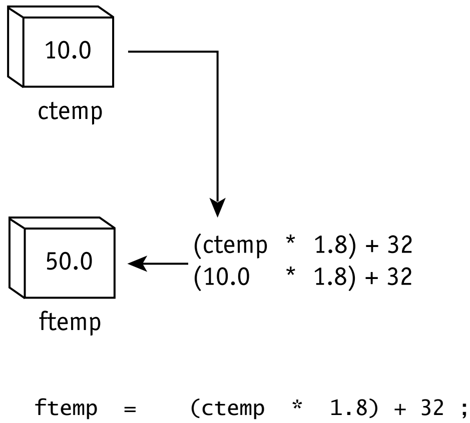
Note Although the number entered in this case was 10, it is stored as 10.0. In purely mathematical terms, 10 and 10.0 are equivalent, but in C++ terms, the notation 10.0 indicates that the value is stored in floating-point format rather than integer format. This turns out to have important consequences.

The next statement performs the actual conversion, using the value stored in ctemp to calculate the value of ftemp:

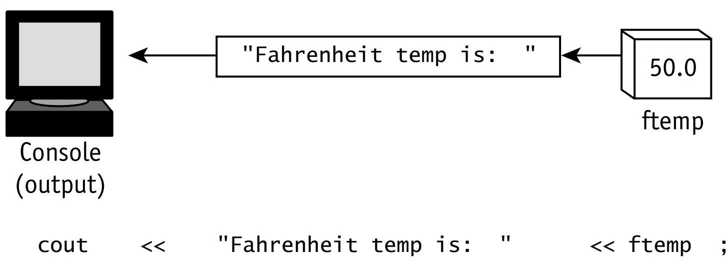
```
ftemp = (ctemp * 1.8) + 32;
```

This statement features an *assignment*: the value on the right side of the equal sign (=) is evaluated and then copied to the variable on the left side. This is one of the most common operations in C++.

Again, assuming that the user input 10, this is how data would flow in the program:



Finally, the program prints the result—in this case, 50.



Optimizing the Program

If you look at the previous example carefully, you might ask yourself, was it really necessary to declare two variables instead of one?

Actually, it wasn't. Welcome to the task of optimization. The following version improves on the first version of the program by getting rid of ftemp and combining the conversion and output steps:

convert3.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    double  ctemp;   // Celsius temperature

    // Prompt and input value of ctemp.

    cout << "Input a Celsius temp and press ENTER: ";
    cin >> ctemp;


    // Convert ctemp and output results.

    cout << "Fahr. temp is: " << (ctemp * 1.8) + 32;
    cout << endl;

    return 0;
}
```

Do you detect a pattern by now? With the simplest programs, the pattern is usually as follows:

1. Declare variables.
2. Get input from the user (after printing a prompt).
3. Perform calculations and output results.

For example, the next program does something different but should look familiar. This program prompts for a number and then prints the square. The statements are similar to those in the previous example but use a different variable (x) and a different calculation.

square.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    double  x = 0.0;

    // Prompt and input value of x.

    cout << "Input a number and press ENTER: ";
    cin >> x;


    // Calculate and output the square.

    cout << "The square is: " << x * x << endl;
    return 0;
}
```

EXERCISES

Exercise 1.3.1. Rewrite the example so it performs the reverse conversion: Input a value into ftemp (Fahrenheit) and convert to ctemp (Celsius). Then print the results. (Hint: The reverse conversion formula is ctemp = (ftemp – 32) / 1.8.)

Exercise 1.3.2. Write the Fahrenheit-to-Celsius program using only one variable, ftemp. This is an optimization of Exercise 1.3.1.

Exercise 1.3.3. Write a program that inputs a value into a variable x and outputs the cube (x * x * x). Make sure the output statement uses the word *cube* rather than *square*.

Exercise 1.3.4. Rewrite the example square.cpp using the variable name num rather than x. Make sure you change the name everywhere “x” appears.

A Word about Variable Names and Keywords

This chapter has featured the variables ctemp, ftemp, and n. Exercise 1.3.4 suggested that you could replace "x" with "num," as long as you do the substitution consistently throughout the program. So "num" is a valid name for a variable as well.

There is an endless variety of variable names I could have used instead. I could, for example, give some variables the names killerRobot or GovernorOfCalifornia.

What variable names are permitted, and what ones are not? You can use any name you want, as long as you follow these rules:

- The first character should be a letter. It cannot be a number. The first character can be an underscore (_), but the C++ library uses that naming convention internally, so it's best to avoid starting a name that way.
- The rest of the name can be a letter, a number, or an underscore (_).
- You must avoid words that already have a special, predefined meaning in C++, such as the keywords.

It isn't necessary to sit down and memorize all the C++ keywords. You need to know only that if you try using a name that conflicts with one of the C++ keywords, the compiler will respond with an error message. In that case, try a different name.

EXERCISE

Exercise 1.3.5. In the following list, which of the words are legal variable names in C++, and which are not? Review the rules just mentioned as needed.

x1

EvilDarkness

PennslyvaniaAve1600

1600PennsylvaniaAve

Bobby_the_Robot

Bobby+the+Robot

whatThe???

amount

count2

count2five

5count

main

main2

Chapter 1 *Summary*

Here are the main points of Chapter 1:

- Creating a program begins with writing C++ source code. This consists of C++ statements, which bear some resemblance to English. (Machine code, by contrast, is completely incomprehensible unless you look up the meaning of each combination of 1s and 0s.) Before the program can be run, it must be translated into machine code, which is all the computer really understands.
- The process of translating C++ statements into machine code is called *compiling*.
- After compiling, the program also has to be linked to standard functions stored in the C++ library. This process is called *linking*. After this step is successfully completed, you have an executable program.
- If you have a development environment, the process of compiling and linking a program (*building*) is automated so you need only press a function key. With Microsoft Visual Studio, press Ctrl+F5 to build programs.
- If you're working with Microsoft Visual Studio, make sure you leave **#include "stdafx"** at the beginning of every program. If you start a project by going through the New Project command, the environment will always put this in for you. Just make sure you don't delete **#include "stdafx"** when pasting code into the environment.

```
#include "stdafx.h"
```

- Simple C++ programs have the following general form:

```
#include <iostream>
using namespace std;

int main()
{
```

```
        Enter_your_statements_here!
        return 0;
    }
```

- To print output, use the **cout** object. For example:

```
cout << "Never fear, C++ is here!";
```

- To print output and advance to the next line, use the **cout** object and send a newline character (**endl**). For example:

```
cout << "Never fear, C++ is here!" << endl;
```

- Most C++ statements are terminated by a semicolon (;). Directives—lines beginning with a pound sign (#)—are a major exception.
- Double slashes (//) indicate a comment; all text to the end of the line is ignored by the compiler itself. But comments can be read by humans who have to maintain the program.
- Before using a variable, you must declare it. For example:

```
double  x;    // Declare x as a floating-pt variable.
```

- Variables that may store a fractional portion should have type **double**. This stands for "double-precision floating point." The single-precision type (**float**) should be used only when storing large amounts of floating-point data on disk.
- To get keyboard input into a variable, you can use the **cin** object. For example:

```
cin >> x;
```

- You can also put data into a variable by using assignment (=). This operation evaluates the expression on the right side of the equal sign (=) and places the value in the variable on the left side. For example:

```
x = y * 2;  // Multiply y times 2, place result in x.
```

Index

Symbols

Numbers

A

G

H

I

T

U

Z

This page intentionally left blank