PEARSON
Addison Wesley

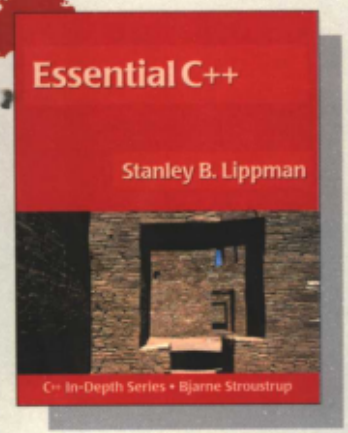
典藏原版书苑

Essential C++

[美] Stanley B. Lippman 著

Essential C++

(英文版)



人民邮电出版社
POSTS & TELECOM PRESS

“拿起这本书，你可以在短时间内熟悉C++。Stanley选择了范围广泛而又复杂的一些主题，把它们的门槛降低到C++新手开发真正的程序时所需要的最基本的层次。本书以实例引导读者学习，这种做法很有效，可帮助读者顺利学完整本书。”

——Steve Vinoski，IONA

对于那些有程序设计经验、但又没有太多时间学习新技术的程序员而言，本书提供了一条快速的学习捷径，通过本书你可以在短时间内把C++应用到工作中。本书重点介绍C++编程过程中极可能遇到的要素，以及可解决实际问题的技术。

Stanley B. Lippman是梦工厂电影动画公司的核心技术小组成员。加入梦工厂之前，他是迪斯尼电影动画公司的主要工程师。当他在Bell实验室的时候，领导了cfront 3.0版本和2.1版本的编译器开发小组。他也是Bjarne Stroustrup领导的贝尔实验室Foundation项目的成员之一。他是*C++ Primer*及*Inside The C++ Object Mode*的作者，以及*C++ Gems*杂志的编辑。他的工作体现在多部动画影片中，包括《钟楼怪人》和《幻想曲2000》。

全书从4个方面来表现C++的本质：procedural（过程化的）、generic（泛型的）、object-based（基于对象的）、object-oriented（面向对象的）。

本书的组织围绕着一系列由浅入深的程序问题，以及用以解决这些问题的语言特性。按照这种方式，读者不但会学到C++的函数和结构，还会学习到它们的设计目的和基本原理。

本书涉及以下关键主题：

- 泛型编程与标准模板库（STL）；
- 基于对象编程与类设计；
- 面向对象编程与类层次设计；
- 函数和类模板的设计与使用；
- 异常处理与运行时类型识别。

此外，两份附录极具价值。附录A提供每章最后所列的练习题的完整解答和详细说明。附录B提供了一份泛型算法快速参考手册（含使用实例）。

这本言简意赅的教程带给你使用C++工作的关键知识，以及未来专业发展的坚实基础。

• 装帧设计：胡平利

ISBN 7-115-15257-8

9 787115 152572 >

人民邮电出版社网址 www.ptpress.com.cn

分类建议：计算机 / 程序设计 / C++

ISBN7-115-15257-8/TP • 5687

定价：32.00 元

Essential C++

（英文版）

[美] Stanley B. Lippman 著

人民邮电出版社

图书在版编目（CIP）数据

Essential C++（英文版）/（美）利普曼（Lippman, S. B）著. —北京：人民邮电出版社，2006.11
（典藏原版书苑）

ISBN 7-115-15257-8

Ⅰ. E… Ⅱ. 利… Ⅲ. C语言—程序设计—英文 Ⅳ. TP312

中国版本图书馆CIP数据核字（2006）第107060号

版 权 声 明

典藏原版书苑

Essential C++（英文版）

◆ 著 [美] Stanley B. Lippman

责任编辑 李 际

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号

邮编 100061 电子函件 315@ptpress.com.cn

网址 http://www.ptpress.com.cn

北京顺义振华印刷厂印刷

新华书店总店北京发行所经销

◆ 开本：800×1000 1/16

印张：17.75

字数：382千字 2006年11月第1版

印数：1－2 500册 2006年11月北京第1次印刷

著作权合同登记号 图字：01-2006-5374号

ISBN 7-115-15257-8/TP・5687

定价：32.00元

读者服务热线：(010)67132705 印装质量热线：(010)67129223

前　言

我写的 *C++ Primer* 全书共 1237 页，而本书只有 276 页，套用拳击术语，这是一本“轻量级”作品。

每个人都会好奇地想知道本书是如何写成的。其中的确有一段故事。

在过去的数年里，我总是缠着迪斯尼动画公司的每一个人，要求他们让我亲身参与一部电影的制作。我缠着导演，让他告诉我幕后真实的情况。我会如此狂热，部分原因是沉湎于好莱坞那令人神往的无穷魅力而难以自拔。我拥有计算机科学学位和艺术硕士学位，而电影工作似乎可以综合发挥我的个人专长。我告诉管理团队，我需要从制片过程中获取经验，以便提供实际有用的工具。身为一名 C++编译器开发者，我一直都是自己最主要的用户之一。而当你是自己软件的主要抱怨者时，你就很难再为自己辩护或觉得受到了不公平的责难。

《幻想曲 2000》（Fantasia 2000）片中有一段火鸟（Firebird）的场景，电脑特效负责人有意邀我加盟制作。为了评估我的水平，他要求我先写个工具，读入某段场景所摄的原始资料，再由此产生可嵌入 Houdini 动画套件中的摄影机节点（camera node）。我当然用 C++顺利把它搞定。他们很喜欢它，我也因此受邀加盟。

有一次，在制作过程中（在此特别感谢 Jinko 和 Chyuan），我被要求以 Perl 重写那个工具。其他的 TD 并非编程高手，仅仅知道 Perl、Tcl 之类的程序语言（TD 是电影工业中的术语，指的是技术导演。我是这部片子的软件 TD，我们还有一位灯光 TD[Dava]，一位模型 TD[Tim]，以及电影特效动画师[Mike, Steve, Tonya]）。而且，我得赶着点，因为我们想要获得一些观念上的实证，而导演（Paul 和 Gaetan）及特效总监（Dave）正等着这个结果，准备呈给公司老板（Peter）。这虽然不是什么紧急要务，可是，你知道的……

这让我感到有些为难。我可以自信地以 C++快速完成，但我不懂 Perl。我想，好吧，我去找本书看看好了，但是这本书不能太厚，起码此刻不能太厚，而且它最好不要告诉我太多东西，虽然我知道我应该全部掌握，不过暂且等等吧。毕竟这只是一次演示：导演们需要一些经过证实的概念，艺术家需要一些东西协助证实其概念，而制片需要的是一天 48 小时。此刻我不需要全世界最棒的 Perl 书，我需要的是一本能正确引导我前进，不会使我偏离正轨过远的小册子。

我找到了 Randal Schwartz 写的 *Learning Perl*，它让我立即上手并进展神速，而且读起来颇具趣味性。不过，就像其他有趣的计算机书籍一样，它也略去了不少值得一读的内容——虽然在那一刻，我并不需要了解所有内容，我只需要让我的 Perl 程序运行起来。

我终于悲哀地意识到，*C++ Primer* 其实无法在人们初学 C++时扮演引导者的角色，它太厚了。尽管如此，我还是认为它是一本让我骄傲的巨著，特别是邀请到了 Josée Lajoie 共同完成。但是，对于想立刻学会 C++程序语言的人来说，这本巨著实在过于庞大复杂。这正是我着手编写 *Essential C++*的原因。

你或许会想，C++又不是 Perl。完全正确！本书也非 *Learning Perl*，它介绍的是如何学习 C++。真正的问题在于，谁能够在用尽千页篇幅之后，犹敢自称介绍了所有的内容呢？

1．精细度。在计算机图形学中，精细度指的是图像被渲染出来的清晰度。画面左上角是一位骑在马

背上的匈奴人，需要描绘出一张看得清楚眼睛的脸、头发、五点钟方向的影子、衣服等。匈奴人的背后——不，不是那块岩石——我们不会以相同的清晰度来渲染这两个图像。同样道理，本书在精细度上降低了相当的程度。依我看，*C++ Primer* 在运算符重载方面的例程讨论可以说极其完备（我敢这么说因为 Josée 是作者）。*C++ Primer* 花了 46 页篇幅加以讨论并附上范例，而本书却仅以 2 页的篇幅带过。

2．语言核心。当我还是 *C++ Report* 杂志的编辑时，我常说，杂志编辑有一半工作时间花在决定哪些内容应该被采用，哪些不被采用。这句话对本书一样成立。本书围绕程序设计过程中所发生的一系列问题，介绍程序语言本身的特性，以此为不同的问题提供解决之道。书中并未涉及多重继承或虚拟继承可解决的问题，所以我也就完全没有讨论这两个主题。然而，为了实现迭代类，我必须引入嵌套类型。类的类型转换运算符很容易被错用，解释起来也很复杂，所以我不打算在书中提到它。诸如此类。欢迎大家针对我对内容的选择以及对语言特性的介绍顺序进行批评指正。我应该对自己的选择负责。

3．范例的数量。*C++ Primer* 有数百页程序代码，其中甚至包括一套面向对象的文本检索系统以及几个完整的类。虽然本书也有程序代码，但数量远不及 *C++ Primer*。为了弥补这项缺憾，我将所有习题解答都放在附录 A 中。诚如我的编辑 Deborah Lafferty 所言："如果你想提高教学速度，唾手可得的解答对强化学习内容极有帮助。"

本书的组织结构

本书由 7 章和两个附录构成。第 1 章通过编写一个交互的小程序介绍 C++语言预定义的内容。这一章涵盖了内建的数据类型、预定义运算符、标准程序库中的 vector 和 string 类、条件语句和循环语句、用于输入和输出的 iostream 程序库。我之所以在本章介绍 vector 和 string 两个类，是因为我提倡读者多多利用它们取代语言内建的数组和 C-style 字符串。

第 2 章解释函数的设计与使用，并针对 C++函数的多种风貌一一介绍，包括 inline 函数、重载函数、标准函数以及函数指针。

第 3 章涵盖所谓的标准模板库（STL）：一组容器类（包括 vector、list、set、map 等）、一组用于容器的泛型算法（包括 sort( )、copy( )、merge( )等）。附录 B 依次列出了被广泛使用的泛型算法，并提供了使用实例。

作为一名 C++程序员，主要任务是提交类以及面向对象的类层次。第 4 章带领你熟悉 C++类机制的设计与使用，为应用程序创建专门的数据类型。以我在梦工厂动画电影公司担任顾问的经验为例，那时候我们设计一些类，用来进行 4 个频道影像合成之类的工作。第 5 章说明如何扩展类，使多个相关的类形成族系，支持面向对象的类层次体系。我们使用继承和动态绑定技术定义影像合成所需的类层次体系，而不只是设计 8 个相互独立的类。

第 6 章重点介绍类模板，类模板是建立类时的一种先行描述，让我们可以将类用到一个（或多个）数据类型或数值并参数化。以 vector 类为例，可能需要将其元素的类型加以参数化。buffer 类的设计不仅要将元素类型参数化，还要将其缓冲区容量参数化。本章围绕二叉树类模板的实现展开讨论。

第 7 章说明如何使用 C++提供的异常处理机制，并将它融入标准程序库所定义的异常类体系中。附录 A 是本书的习题解答。附录 B 提供被广泛运用的一些泛型算法的相关讨论与使用实例。

关于程序代码

本书的所有程序以及习题解答中的完整程序代码皆可在网上获得，也可以在 Addison Wesley Longman 的网站（www.awl.com/cseng/titles/0-201-48518-4）或我的个人主页（www.objectwrite.com）中获得。所有

程序皆在 Visual C++ 5.0 环境中以 Intel C++编译器测试通过，并且也在 Visual C++ 6.0 环境中以 Microsoft C++编译器测试通过。你或许需要稍微修改程序代码，才能在自己的系统上编译成功。如果你做了任何修改，请将修改结果寄一份给我（slippman@objectwrite.com），我会将它们附上你的大名，列于习题解答程序代码中。注意，本书并未给出所有程序代码。

致谢

在这里我要特别感谢 *C++ Primer* 的合著者 Josée Lajoie。不仅因为她为本书初稿提供了许多独到的见解，更因为她在背后不断地鼓舞我。我也要特别感谢 Dava Slayton 彻底检查了本书的正文内容与程序范例。Steve Vinoski 则以同情但坚决的口吻为本书初稿提供了许多宝贵意见。

特别感谢 Addison Wesley 编辑小组的以下成员：本书编辑 Deborah Lafferty，他从头到尾支持这个想法；文字编辑 Besty Hardinger，他对本书文字的可读性贡献最大；产品经理 John Fuller，他带领我们把一堆文稿转化为一本完整的书。

在本书的撰写过程中，我同时还担任独立顾问工作，必须兼顾 *Essential C++*和客户之间的事务。感谢我的客户对我如此地体谅和宽容。我要感谢 Colin Lipworth、Edwin Leonard、Kenneth Meyer，因为他们的耐心与信赖，本书才得以完成。

更多信息

我要推荐 C++书籍中最好的两本，那便是 Lippman 和 Lajoie 合著的 *C++ Primer*，以及 Stroustrup 著的 *The C++ Programming Language*。我会在本书各部分中提供其他更深入的参考书目。以下便是本书的参考书目（你可以在 *C++ Primer* 和 *The C++ Programming Language* 中找到更广泛的参考文献）。

[LIPPMAN98] Lippman, Stanley and Josée Lajoie, *C++ Primer*, 3rd Edition, Addison Wesley Longman, Inc., Reading, MA(1998) ISBN 0-201-82470-1.

[LIPPMAN96a] Lippman, Stanley, *Inside the C++ Object Model*, Addison Wesley Longman, Inc., Reading, MA(1996) ISBN 0-201-83454-5.

[LIPPMAN96b] Lippman, Stanley, Editor, *C++ Gems*, a SIGS Books imprint, Cambridge University Press, Cambridge, England (1996) ISBN 0-13570581-9.

[STROUSTRUP97] Stroustrup, Bjarne, *The C++ Programming Language*, 3rd Edition, Addison Wesley Longman, Inc., Reading, MA (1997) ISBN 0-201-88954-4.

[STROUSTRUP99] Sutter, Herb, *Exceptional C++*, Addison Wesley Longman, Inc., Reading, MA (2000) ISNB 0-201-61562-2.

字体习惯

本书文字字体为 10.5 磅 Palatino 字体。程序代码和语言关键词为 8.5 磅 `lucida` 字体。如果书中出现的标识符后面紧接着 C++的函数调用运算符（也就是一对小括号()），即代表某个函数名称。例如，`foo` 代表程序中的某个对象，`bar( )`代表程序中的函数。各个类的名称以 Palatino 字体显示。

Contents

1

Basic C++ Programming

In this chapter, we evolve a small program to exercise the fundamental components of the C++ language. These components consist of the following:

1. A small set of data types: Boolean, character, integer, and floating point.
2. A set of arithmetic, relational, and logical operators to manipulate these types. These include not only the usual suspects, such as addition, equality, less than, and assignment, but also the less conventional increment, conditional, and compound assignment operators.
3. A set of conditional branch and looping statements, such as the `if` statement and `while` loop, to alter the control flow of our program.
4. A small number of compound types, such as a pointer and an array. These allow us, respectively, to refer indirectly to an existing object and to define a collection of elements of a single type.
5. A standard library of common programming abstractions, such as a string and a vector.

1.1 How to Write a C++ Program

We've been asked to write a simple program to write a message to the user's terminal asking her to type in her name. Then we read the name she enters, store the name so that we can use it later, and, finally, greet the user by name.

OK, so where do we start? We start in the same place every C++ program starts — in a function called `main()`. `main()` is a user-implemented function of the following general form:

```
int main()
{
   // our program code goes here
}
```

`int` is a C++ language keyword. *Keywords* are predefined names given special meaning within the language. `int` represents a built-in integer data type. (I have much more to say about data types in the next section.)

A *function* is an independent code sequence that performs some computation. It consists of four parts: the return type, the function name, the parameter list, and the function body. Let's briefly look at each part in turn.

The *return type* of the function usually represents the result of the computation. `main()` has an integer return type. The value returned by `main()` indicates whether our program is successful. By convention, `main()` returns 0 to indicate success. A nonzero return value indicates something went wrong.

The *name* of a function is chosen by the programmer and ideally should give some sense of what the function does. `min()` and `sort()`, for example, are pretty good function names. `f()` and `g()` are not as good. Why? Because they are less informative as to what the functions do.

`main` is not a language keyword. The compilation system that executes our C++ programs, however, expects a `main()` function to be defined. If we forget to provide one, our program will not run.

The *parameter list* of a function is enclosed in parentheses and is placed after the name of the function. An empty parameter list, such as that of `main()`, indicates that the function accepts no parameters.

The parameter list is typically a comma-separated list of types that the user can pass to the function when the function is executed. (We say that the user has *called*, or *invoked*, a function.) For example, if we write a function `min()` to return the smaller of two values, its parameter list would identify the types of the two values we want to compare. A `min()` function to compare two integer values might be defined as follows:

```
int min( int val1, int val2 )
{
   // the program code goes here ...
}
```

The *body* of the function is enclosed in curly braces ({}). It holds the code sequence that provides the computation of the function. The double forward slash (//) represents a comment, a programmer's annotation on some aspect of the code. It is intended for readers of the program and is discarded during compilation. Everything following the double forward slash to the end of the line is treated as a comment.

Our first task is to write a message to the user's terminal. Input and output are not a predefined part of the C++ language. Rather, they are supported by an *object-oriented class hierarchy* implemented in C++ and provided as part of the C++ standard library.

A class is a user-defined data type. The class mechanism is a method of adding to the data types recognized by our program. An object-oriented class hierarchy defines a family of related class types, such as terminal and file input, terminal and file output,

and so on. (We have a lot more to say about classes and object-oriented programming throughout this text.)

C++ predefines a small set of fundamental data types: Boolean, character, integer, and floating point. Although these provide a foundation for all our programming, they are not the focus of our programs. A camera, for example, must have a location in space, which is generally represented by three floating point numbers. A camera also has a viewing orientation, which is also represented by three floating point numbers. There is usually an aspect ratio describing the ratio of the camera viewing width to height. This is represented by a single floating point number.

On the most primitive level, that is, a camera is represented as seven floating point numbers, six of which form two x,y,z coordinate tuples. Programming at this low level requires that we shift our thinking back and forth from the manipulation of the camera abstraction to the corresponding manipulation of the seven floating point values that represent the camera in our program.

The class mechanism allows us to add layers of type abstraction to our programs. For example, we can define a Point3d class to represent location and orientation in space. Similarly, we can define a Camera class containing two Point3d class objects and a floating point value. We're still representing a camera by seven floating point values. The difference is that in our programming we are now directly manipulating the Camera class rather than seven floating point values.

The definition of a class is typically broken into two parts, each represented by a separate file: a *header file* that provides a declaration of the operations supported by the class, and a *program text file* that contains the implementation of those operations.

To use a class, we include its header file within our program. The header file makes the class known to the program. The standard C++ input/output library is called the *iostream* library. It consists of a collection of related classes supporting input and output to the user's terminal and to files. To use the iostream class library, we must include its associated header file:

```
#include <iostream>
```

To write to the user's terminal, we use a predefined class object named `cout` (pronounced *see out*). We direct the data we wish `cout` to write using the output operator (<<), as follows:

```
cout << "Please enter your first name: ";
```

This represents a C++ program *statement*, the smallest independent unit of a C++ program. It is analogous to a sentence in a natural language. A statement is terminated by a semicolon. Our output statement writes the string literal (marked by double quotation marks) onto the user's terminal. The quotation marks identify the string; they are not displayed on the terminal. The user sees

```
Please enter your first name:
```

Our next task is to read the user's input. Before we can read the name the user types, we must define an object in which to store the information. We define an object by specifying the data type of the object and giving it a name. We've already seen one data type: `int`. That's hardly a useful way of storing someone's name, however! A more appropriate data type in this case is the standard library string class:

```
string user_name;
```

This defines `user_name` as an object of the string class. The definition, oddly enough, is called a *declaration statement*. This statement won't be accepted, however, unless we first make the string class known to the program. We do this by including the string class header file:

```
#include <string>
```

To read input from the user's terminal, we use a predefined class object named `cin` (pronounced *see in*). We use the input operator (`>>`) to direct `cin` to read data from the user's terminal into an object of the appropriate type:

```
cin >> user_name;
```

The output and input sequence would appear as follows on the user's terminal. (The user's input is highlighted in bold.)

```
Please enter your first name: anna
```

All we've left to do now is to greet the user by name. We want our output to look like this:

```
Hello, anna ... and goodbye!
```

I know, that's not much of a greeting. Still, this is only the first chapter. We'll get a bit more inventive before the end of the book.

To generate our greeting , our first step is to advance the output to the next line. We do this by writing a newline character literal to `cout`:

```
cout << '\n';
```

A character literal is marked by a pair of single quotation marks. There are two primary flavors of character literals: printing characters such as the alphabet (`'a'`, `'A'`, and so on), numbers, and punctuation marks (`';'`, `'-'`, and so on), and nonprinting characters such as a newline (`'\n'`) or tab (`'\t'`). Because there is no literal representation of nonprinting characters, the most common instances, such as the newline and tab, are represented by special two-character sequences.

Now that we've advanced to the next line, we want to generate our `Hello`:

```
cout << "Hello, ";
```

Next, we need to output the name of the user. That's stored in our string object, `user_name`. How do we do that? Just the same as with the other types:

```
cout << user_name;
```

Finally, we finish our greeting by saying goodbye (notice that a string literal can be made up of both printing and nonprinting characters):

```
cout << " ... and goodbye!\n";
```

In general, all the built-in types are output in the same way — that is, by placing the value on the right-hand side of the output operator. For example,

```
cout << "3 + 4 = ";
cout << 3 + 4;
cout << '\n';
```

generates the following output:

```
3 + 4 = 7
```

As we define new class types for use in our applications, we also provide an instance of the output operator for each class. (We see how to do this in Chapter 4.) This allows users of our class to output individual class objects in exactly the same way as the built-in types.

Rather than write successive output statements on separate lines, we can concatenate them into one compound output statement:

```
cout << '\n'
     << "Hello, "
     << user_name
     << " ... and goodbye!\n";
```

Finally, we can explicitly end `main()` with the use of a return statement:

```
return 0;
```

`return` is a C++ keyword. The expression following `return`, in this case 0, represents the result value of the function. Recall that a return value of 0 from `main()` indicates that the program has executed successfully.[1]

Putting the pieces together, here is our first complete C++ program:

```
#include <iostream>
#include <string>
using namespace std; // haven't explained this yet ...

int main()
{
      string user_name;
      cout << "Please enter your first name: ";
      cin >> user_name;
      cout << '\n'
           << "Hello, "
```

[1] If we don't place an explicit `return` statement at the end of `main()`, a `return 0;` statement is inserted automatically. In the program examples in this book, I do not place an explicit `return` statement.

```
            << user_name
            << " ... and goodbye!\n";

    return 0;
}
```

When compiled and executed, this code produces the following output (my input is highlighted in bold):

```
Please enter your first name: anna
Hello, anna ... and goodbye!
```

There is one statement I haven't explained:

```
using namespace std;
```

Let's see if I can explain this without scaring you off. (A deep breath is recommended at this point!) Both `using` and `namespace` are C++ keywords. `std` is the name of the standard library namespace. Everything provided within the standard library (such as the string class and the iostream class objects `cout` and `cin`) is encapsulated within the `std` namespace. Of course, your next question is, what is a namespace?

A *namespace* is a method of packaging library names so that they can be introduced within a user's program environment without also introducing name clashes. (A *name clash* occurs when there are two entities that have the same name in an application so that the program cannot distinguish between the two. When this happens, the program cannot run until the name clash is resolved.) Namespaces are a way of fencing in the visibility of names.

To use the string class and the iostream class objects `cin` and `cout` within our program, we must not only include the string and iostream header files but also make the names within the `std` namespace visible. The *using directive*

```
using namespace std;
```

is the simplest method of making names within a namespace visible. (To read about namespaces in more detail, check out either Section 8.5 of [LIPPMAN98] or Section 8.2 of [STROUSTRUP97].)

Exercise 1.1

Enter the `main()` program, shown earlier. Either type it in directly or download the program; see the Preface for how to acquire the source programs and solutions to exercises. Compile and execute the program on your system.

Exercise 1.2

Comment out the string header file:

```
// #include <string>
```

Now recompile the program. What happens? Now restore the string header and comment out

```
//using namespace std;
```

What happens?

Exercise 1.3

Change the name of `main()` to `my_main()` and recompile the program. What happens?

Exercise 1.4

Try to extend the program: (1) Ask the user to enter both a first and last name and (2) modify the output to write out both names.

1.2 Defining and Initializing a Data Object

Now that we have the user's attention, let's challenge her to a quiz. We display two numbers representing a numerical sequence and then request our user to identify the next value in the sequence. For example,

```
The values 2,3 form two consecutive
    elements of a numerical sequence.
What is the next value?
```

These values are the third and fourth elements of the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, and so on. A Fibonacci sequence begins with the first two elements set to 1. Each subsequent element is the sum of its two preceding elements. (In Chapter 2 we write a function to calculate the elements.)

If the user enters 5, we congratulate her and ask whether she would like to try another numerical sequence. Any other entered value is incorrect, and we ask the user whether she would like to guess again.

To add interest to the program, we keep a running score based on the number of correct answers divided by the number of guesses.

Our program needs at least five objects: the string class object to hold the name of the user; three integer objects to hold, in turn, the user's guess, the number of guesses, and the number of correct guesses; and a floating point object to hold the user's score.

To define a data object, we must both name it and provide it with a data type. The name can be any combination of letters, numbers, and the underscore. Letters are case-sensitive. Each one of the names `user_name`, `User_name`, `uSeR_nAmE`, and `user_Name` refers to a distinct object.

A name cannot begin with a number. For example, `1_name` is illegal but `name_1` is OK. Also, a name must not match a language keyword exactly. For example, `delete` is a language keyword, and so we can't use it for an entity in our program. (This explains

why the operation to remove a character from the string class is `erase()` and not `delete()`.)

Each object must be of a particular data type. The name of the object allows us to refer to it directly. The data type determines the range of values the object can hold and the amount of memory that must be allocated to hold those values.

We saw the definition of `user_name` in the preceding section. We reuse the same definition in our new program:

```
#include <string>
string user_name;
```

A class is a programmer-defined data type. C++ also provides a set of built-in data types: Boolean, integer, floating point, and character. A keyword is associated with each one to allow us to specify the data type. For example, to store the value entered by the user, we define an integer data object:

```
int usr_val;
```

`int` is a language keyword identifying `usr_val` as a data object of integer type. Both the number of guesses a user makes and the number of correct guesses are also integer objects. The difference here is that we wish to set both of them to an initial value of 0. We can define each on a separate line:

```
int num_tries = 0;
int num_right = 0;
```

Or we can define them in a single comma-separated declaration statement:

```
int num_tries = 0, num_right = 0;
```

In general, it is a good idea to initialize data objects even if the value simply indicates that the object has no useful value as yet. I didn't initialize `usr_val` because its value is set directly from the user's input before the program makes any use of the object.

An alternative initialization syntax, called a *constructor syntax*, is

```
int num_tries( 0 );
```

I know. Why are there two initialization syntaxes? And, worse, why am I telling you this now? Well, let's see whether the following explanation satisfies either or both questions.

The use of the assignment operator (=) for initialization is inherited from the C language. It works well with the data objects of the built-in types and for class objects that can be initialized with a single value, such as the string class:

```
string sequence_name = "Fibonacci";
```

It does not work well with class objects that require multiple initial values, such as the standard library complex number class, which can take two initial values: one for its real component and one for its imaginary component. The alternative constructor