深入 C++ 系列

《C++ 沉思录》作者作品
美国斯坦福大学 C++ 教材

Accelerated C++ 中文版

Practical Programming by Example

[美] Andrew Koenig　Barbara E. Moo 著
覃剑锋　柯晓江　蓝图 等 译
王昕 校

Addison Wesley

中国电力出版社
www.infopower.com.cn

深入 C++ 系列

Accelerated C++ 中文版

Practical Programming by Example

[美] Andrew Koenig　Barbara E. Moo 著
覃剑锋 柯晓江 蓝图 等 译
王昕 校

中国电力出版社

Accelerated C++ (ISBN 0-201-70353-X)
Andrew Koenig Barbara E. Moo

图书在版编目（CIP）数据
Accelerated C++中文版 /（美）克尼格著；覃剑锋等译.
北京：中国电力出版社，2003
ISBN 7-5083-1819-6
Ⅰ.A... Ⅱ.①克...②覃... Ⅲ.C 语言－程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 100502 号

丛 书 名：深入C++系列
书　　名：Accelerated C++中文版
编　　著：（美）Andrew Koenig Barbara E. Moo
翻　　译：覃剑锋 柯晓江 蓝图 等
审　　校：王昕
责任编辑：陈维宁
出版发行：中国电力出版社
地址：北京市三里河路6号 邮政编码：100044
电话：（010）88515918 传　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092 1/16 印　张：22 字　数：482 千字
书　　号：ISBN 7-5083-1819-6
版　　次：2003 年 12 月北京第 1 版 2003 年 12 月第 1 次印刷
定　　价：39.50 元

前　言

教授 C++编程的一种新方法

我们假设，读者们希望迅速地掌握编写实用 C++程序的方法。因此，我们会首先解释 C++的最有用的部分。如果我们这样去理解的话，那这个策略看上去似乎是显而易见的，不过，它却有着根本的含意——那就是，即使 C++是以 C 为基础的，但是我们也并不会从 C 的教学开始。相反，我们从一开始就使用了高级数据结构，而且只会在以后才去解释这些数据结构所依赖的基础。这个方法可以让你马上开始编写地道的 C++程序。

从另一个角度来看，我们的方法也是很独特的：我们集中注意力来解决问题，而不是专门去探究语言和库的特征。当然，我们也会解释这些特征，但这样做的目的是为程序提供支持，而不是用程序来作为演示特征的工具。

因为本书是教授 C++程序设计而不单单是讲解语言特征的，所以对那些已经具备一定 C++基础并想以更自然、更高效的风格使用这门语言的读者来说，它尤为有用。C++的初学者会很注重语言技巧的学习，但是他们却常常不懂得如何运用它们来解决日常的问题。

我们的方法对初学者及熟练的程序员同样适用

在过去的每个暑假中我们都会在斯坦福大学开设为期一周的 C++强化教程。一开始，我们在教学中采用了传统的方法：假定学生已经掌握 C 语言，于是我们的教学从类的定义方法开始，然后会系统地过渡到语言的其他方面。我们发现，学生们会在开始的两三天内感到困惑并且会出现挫败感——直到他们掌握的知识足以让他们编写出实用的程序了，这种现象才会消失。而一旦到达了这种程度，他们就可以很轻松地继续学下去。

当我们接触到一种对崭新的标准库提供足够支持的 C++系统环境的时候，我们就对课程进行了彻底的更新。在新的课程中，我们从一开始就使用了标准库，同时将注意力集中到用来编写实用的程序上。而且，只是在学生掌握了足以让他们高效地使用各种语言细节的知识以后，我们才对语言的细节进行深入的探讨。

结果是很戏剧化的：一天之后，我们的学生就能在课堂上编写出在旧教程中要花费他们大半周时间的程序。而且，他们的挫败感也消失得无影无踪。

抽象

我们的方法之所以适用，是因为 C++（以及我们对它的了解）已经渐趋成熟了。这种成

熟可以让我们忽略许多为早期的C++程序和程序员所依赖的低层次的概念。

这种允许忽略细节的能力是成熟技术的一个特征。例如，早期的汽车是经常会出故障的，因此每一个司机都迫于无奈地变成了业余的技工。那时候，在不懂得如何解决驾驶中的突发问题的情况下，人们是不敢贸然出去驾驶的。今天的司机无需掌握详细的工程知识就可以使用汽车来进行运输了。当然，他们可能会出于其他目的而去学习工程学的细节，但那又是另外一回事了。

我们把抽象定义为可供选择的忽略(把注意力集中于与手头上的任务相关的概念而忽略了其他所有的枝节)我们认为这是现代程序设计中最重要的方法。编写一个成功的程序的关键在于是否清楚问题的哪些部分应给予考虑，哪些部分应该被忽略。每一种程序设计语言都提供了工具来让我们创造有用的抽象，而每个成功的程序员都应该懂得如何使用这些工具。

我们认为抽象是非常实用的，因此在本书中到处都可以见到抽象。当然，我们通常并不直接称之为抽象，因为它们是以各种各样的形式出现的。相反，我们提到了函数、数据结构、类以及继承——所有这些都是抽象。我们不但是提及它们，而且在书中我们还会常常使用它们。

如果我们能够很好地去设计和选择抽象，那么我们就有理由相信，即使在不了解其所有细节的情况下，我们也可以正确地使用它们。我们无需成为机械工程师就可以驾驶汽车，同样的道理，我们在使用C++之前也无需了解关于C++运作的所有细节。

覆盖范围

如果你是以严谨的态度去看待C++程序设计的话，那你就有必要去掌握我们在本书中所介绍的所有知识点——尽管这本书并没有向你介绍你需要知道的一切知识。

这句话听起来有些绕口，但却并不矛盾。没有哪一本类似厚度的书能够覆盖你需要了解的关于C++的一切知识，因为不同的程序员和应用需要不同的知识。因此，任何一本覆盖了C++的所有知识的书籍——例如Stroustrup所著的《The C++ Programming Language》(Addison-Wesley，2000)——都会无可避免地向你介绍许多你无需了解的东西。这是因为，即使你不需要它们，也自有其他人会有这个需要。

另一方面，C++的许多部分是十分重要的。因此，如果想取得高效率的话，我们就必须切实地掌握它们。我们会把注意力集中在这些重要的部分。而仅仅应用本书提供的信息来编写各种各样的实用程序是完全有可能的。事实上，本书的一个复审者(他是一个用C++编写的大型商业系统的主程序员)告诉我们，这本书基本上涵盖了他在工作中所要用到所有的工具。

拥有了这些工具，我们就可以编写真正的C++程序了——而不仅仅是编写具有C语言或其他任何语言风格的程序。掌握了这本书所介绍的知识之后，我们就能很轻松地继续往下学我们所希望学到的知识，而且我们也因此而获得了一个科学的学习方法。在业余的望远镜制造者群体中流传着一个这样的说法，就是先制造一个3英寸的镜片然后造出6英寸的镜片比从头开始制造一个6英寸的镜片更容易——对C++来说也是同样的道理。

在这本书中，我们仅仅覆盖了标准C++而忽略了其他专门的延伸。这种做法有其独有的

优势，也就是说，我们教你编写的程序可以在任何环境下运行。不过，这也表明了，我们不会探讨如何去编写运行在窗口环境下的程序，因为这样的程序将会无可避免地与特定的环境或与特定的厂商密切联系在一起。如果你想编写仅仅在特定的环境下运行的程序，那你就必须通过其他途径去学习具体的编写方法——但千万不要就这样把这本书合上了！这是因为，我们在这本书中介绍的方法是普遍适用的，以后你可以在任何的环境中使用在这里学到的任何知识。当然，如果你希望了解 GUI 的应用原理的话，那么你应该继续阅读一些关于这一方面的书籍——但是，请首先阅读这一本。

给熟练的 C 和 C++程序员的一点提醒

在学习一门新的程序设计语言的时候，我们可能会不自觉地用类似于我们已经了解的语言的风格去编写程序。我们的教学方法从一开始就使用了来自 C++标准库的高级抽象，这样就可以避免这种不自觉了。如果你已经是一个熟练的 C 或 C++程序员，那么你会从这个方法中得到一些好消息和一些坏消息——而实际上，好消息、坏消息都只不过是同样的消息罢了。

这些消息就是，在我们介绍 C++的时候，你可能会感觉到很惊奇——因为你所掌握的知识对你的学习来说好像用处并不大。一开始，你需要学习的知识将会比你意料中的多得多（这是坏消息），但你的学习效率将会比你预期的要高很多（这是好消息）。特别地，如果你已经对 C++有所了解，那么，之前你首先学习的很可能会是 C 编程，这就意味着你的 C++程序风格是建立在 C 的基础上的。这种做法本身并没有错，但是，我们的方法是如此的与众不同，以至于我们认为，我们会让你看到你从未见过的 C++的另一面。

当然，许多语法细节都是相似的，不过它们仅仅是细节。我们处理重要概念时所采用的顺序很可能会跟你之前接触过的完全不同。例如，直到第 10 章我们才会提到指针和数组，另外，我们甚至根本不会对你们可能非常熟悉的 printf 和 malloc 进行讨论。另一方面，我们会在第 1 章就开始讨论标准库的 string 类。我们所说的要采用一种全新的方法是名副其实的！

本书的结构

你有可能会发现，把本书分成两部分来考虑会更方便一点。第一部分是从开始到第 7 章，在这一部分，我们将把注意力集中于使用标准库抽象的程序；第二部分从第 8 章开始，我们会在这部分定义属于我们自己的抽象。

对库进行介绍首先就是一种不寻常的想法，但我们认为这种想法是正确的。C++语言的许多部分（尤其是那些较难的部分）在很大的程度上是为了库作者的利益而存在的，库的使用者根本无需了解语言的这些部分。因此，我们一直到本书的第二部分才提及这些部分，这样的话，很快我们就可以编写实用的 C++程序了。而如果使用的是一种更为传统的方法，那我们可能还要过很久才能开始编写程序。

一旦掌握了库的使用方法，你就可以开始学习那些低级工具并且可以尝试使用这些工具来编写你自己的库了，顺便提一下，库就是以这些工具为基础而建立起来的。另外，对于如何让

一个库变得更加有用以及何时应该避免全部重写新的代码这两点，你也将会有一个感性的认识。

虽然这本书的厚度比许多 C++书籍要薄，但是，在书中我们尽量将每一个重要的概念使用至少两次以上，而关键概念的使用次数则更多。因此，我们在本书的许多部分中会提到其他的部分。在我们提及其他部分的时候，我们采用诸如 § 39.4.3 这样的形式，§ 39.4.3 指的是第 39.4.3 小节——或者说，如果这本书有这么多的小节的话，它至少会有此意义。在第一次对概念进行解释时，我们都会用粗体来标明这个概念，因为这样做可以方便读者的查找，而且这样还可以吸引读者的注意力，让读者把它当作一个重点来看待。

本书的每一章（除了最后一章）都会有一部分叫做“小结”的内容。这一部分内容是为两个目的服务的：它们可以让你加深对本章所介绍的概念的记忆；同时，它们也涵盖了额外的一些相关的知识——我们认为你迟早会需要了解这些知识。建议读者们在初次阅读时跳过这些内容，在日后有需要时再来参考它们。

本书的两个附录在细节的层次上概述并解释了语言和库的重要组成部分，我们希望它们会对你编写程序有所帮助。

让本书物尽其用

每一本关于程序设计的书都会包含有程序，这一本也不例外。为了理解程序是如何工作的，就必须在计算机上运行它们。这样的计算机到处都有，而且新的计算机也会不断出现——这句话的意思是说，到你读到这句话为止，我们提到的任何关于它们的信息都可能会是不准确的。因此，如果你还不知道怎样编译和运行 C++程序的话，那么请访问站点 http://www.acceleratedcpp.com 并参阅我们在那里发布的信息。我们会不断更新此站点的内容，为它添加关于 C++程序运行技巧的信息和建议。我们在这个网站中也提供了一些机器可读的示例程序版本和一些或许会让你感兴趣的其他信息。

致谢

我们谨对以下的人们表示我们的谢意，因为没有他们就不可能会有这本书的诞生。本书的成功在很大程度上要归功于我们的复审人员：Robert Berger, Dag Brück, Adam Buchsbaum,Stenphen Clamage, John Kalb, Jeffrey Oldham, David Slayton, Bjarne Stroustrup, Albert Tenbusch, Bruce Tetelman 和 Clovis Tondo。许多来自 Addison-Wesley 的工作人员参与了本书的出版工作；我们所知道的有：Tyrrell Albaugh, Bunny Ames, Mike Hendrickson, Deborah Lafferty, Cathy Ohala 以及 Simone Payment 等等。Alexander Tsiris 核对了 § 13.2.2 中的希腊词源。最后要说的是，开始高级编程的想法已经在我们脑海中萦绕了好几年了，这是受到数以百计听过我们课程的学生和成千上万参与我们讨论的人们的激励而产生的。

Andrew Koenig
Barbara E.Moo

Gillette，New Jersey

目 录

第 0 章

开始学习 C++

让我们从一个小的 C++程序开始我们的学习：

```
//一个小的 C++程序
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

程序员经常把这样的一个小程序称为 Hello, world!程序。尽管这个程序很小，但是，在往下阅读之前你还是应该抽点时间出来在你的计算机上编译和运行一下它。这个程序会在标准输出上显示 Hello, world!。一个典型的标准输出是显示在屏幕上的一个窗口。如果碰到麻烦的话，可以找那些已经了解 C++的人来寻求帮助，也可以咨询我们的站点 http://www.acceleratedcpp.com。

由于它的简短，这个程序还是很有用的。如果你对如此简单的一个程序都有问题的话，那么最可能的原因就是本书中存在着明显的印刷错误或者你还没有掌握编译器实现的使用方法。此外，透彻地了解哪怕是一个如此简单的程序也会教给你许多出乎你意料之外的 C++基础原理。为了让我们的理解更加深入，让我们逐行地来分析这个程序吧。

0.1 注释

程序的第一行是：

```
//一个小的 C++程序
```

字符//表示一段**注释**的开始，以这种方式开始的注释将会一直延伸至该行结束。编译器在编译时会忽略掉注释；它们的用途是为阅读该程序的人解释程序。

0.2 #include 指令

C++中的许多基本工具，例如输入/输出，都不属于**语言核心**，而是**标准库**的一部分。这个差别是很重要的——因为语言核心对所有的 C++程序来说都是可用的，但是，在使用标准库时，我们必须明确指定我们所希望使用的那部分标准库。

#include 指令就是这样的一种工具。它一般出现在程序的开头。在我们的程序中，所用到的标准库部分仅仅是输入/输出。我们通过下面的语句来请求使用它：

```
#include <iostream>
```

名称 iostream 表示对顺序或流输入/输出的支持，不过它不支持随机存储或图形输入/输出。由于 iostream 出现在#include 指令中，而且被尖括号（<和>）括起，所以它就代表 C++库的**标准头文件**。

C++标准文档并没有告诉我们**标准头文件**是什么，但是它明确地定义了每一个头文件的名称和行为。在包含了**标准头文件**之后，程序就可以使用相关的库所提供的功能了——我们并不需要去关心它的实现，这是编译器实现所需考虑的事情。

0.3 主函数 main

函数（function）是一段具有名称的程序，程序的其他部分可以**调用**函数或使函数运行。每个 C++程序都必须包含一个名为 main 的函数。当我们请求 C++实现运行程序时，它就会调用这个函数来响应我们的请求。

main 函数要返回一个整数类型的值作为其结果，这样它就可以告知编译器是否运行成功。零值表示成功；任何其他的值都意味着程序运行有问题。因此，我们用

```
int main()
```

来表示我们定义了一个名为 main、返回值类型为 int 的函数。在这里，int 是核心语言用来描述整数的名称。main 后面的括号括住了函数从编译器中接收到的参数。在这个特殊的例子中，main 函数并没有参数，因此括号内没有任何东西。在§10.4 中我们将会看到 main 函数参数的使用方法。

0.4 花括号

紧接着上面所说的括号，让我们继续分析 main 函数的定义。在 main 函数中，括号的后面是一连串用花括号括住的语句：

```
int main()
{               //左花括号
                //语句放此处
```

```
}                    //右花括号
```

在 C++中，花括号告诉编译器把出现在它们之间的所有内容当作一个单元来处理。在本例中，左花括号标明了我们的主函数语句的开始，右花括号则标明了它的结束。换句话说，花括号表明，在它们之间的所有语句都隶属于同一个函数。

如我们在这个函数中见到的那样，如果在花括号中有两条或更多的语句，那么编译器就会按照这些语句出现的先后顺序来执行它们。

0.5 使用标准库进行输出

花括号内的第一条语句执行了程序的实际工作。

```
std::cout << "Hello, World!" << std::endl;
```

这条语句首先使用了标准库的输出运算符<<来把 Hello, world!写到标准输出中，然后它写入了 std:endl 的值。

名称之前的 std::表明了这个名称是一个名为 std 的名字空间的一部分。名字空间是一个相关名称的集合；标准库使用 std 来包含所有由它定义的名称。例如，标准头文件 iostream 定义了名称 cout 和 endl，这些名称实际上也就是我们所使用的 std::cout 和 std::endl。

名称 std::cout 指**标准输出流**，标准输出流是所有 C++实现用来进行普通的程序输出的工具。在窗口操作系统环境下的一个典型 C++实现中，在程序运行的时候，系统环境会把程序和一个窗口联系起来，而 std::cout 就指示了这个窗口。在这样的系统环境下，写到 std::cout 中的输出将会出现在相应的窗口中。

我们用值 std::endl 来作为当前输出语句行的结束，如果程序需要产生更多的输出，接下来的输出就会在新的一行中出现。

0.6 返回语句

返回（return）语句，例如

```
return 0;
```

会在其出现的位置终止函数的执行，并把一个值返回给调用这个函数的程序，返回值出现在 return 和分号之间（本例中为 0）。返回值的类型必须和函数声明的返回类型一致。就 main 函数而言，返回类型是 int，接收 main 的返回值的程序是 C++实现本身。因此，main 函数中的 return 语句必须包含一个整数值的表达式，这个表达式会把返回值传递给实现。

当然，我们有可能会在多个位置上合理地终止程序，这样的程序可能会有多条的 return 语句。如果函数定义保证了函数会返回一个特定类型的值，那么函数中的每条 return 语句都必须返回一个适当类型的值。

0.7 一些较为深入的观察

这个程序使用了两个在C++中到处可见的概念：表达式和作用域。随着本书内容的进展，我们会对这些概念进行更为深入的讨论，但是，在这里我们有必要对某些基础知识进行初步的了解。

表达式会让编译器对某些事物进行运算。运算会产生一个**结果**，同时还有可能会具有**副作用**——也就是说，副作用不是结果的直接部分，但它会影响程序或系统环境的状态。例如，3+4是一个表达式，它产生的结果是7，这个运算并没有副作用。而

```
std:cout<<"Hello, world!"<<std:endl
```

就是一个具有副作用的表达式，它会把Hello,world!写进标准输出流而且会结束当前行。

表达式包含有操作数和运算符，操作数和运算符都能以多种形式出现。在我们的Hello,world!表达式中，两个<<符号是运算符，std::cout、"Hello,world!"和std::endl则是操作数。

每一个操作数都具有一个**类型**。以后我们会进一步讨论类型，但是，从本质上来看，类型表示的是一种数据结构和对此数据结构的合理操作。运算符的效果取决于它的操作数的类型。

类型通常会有名称。例如，核心语言定义了int来作为一种类型的名称，这种类型代表的是整数。库定义了std::ostream来作为提供了基于流的输出的类型。在我们的程序中，std::cout的类型就是std::ostream。

运算符<<具有两个操作数，然而我们却使用了两个<<运算符和三个操作数。为什么会这样呢？答案是：<<是**左结合**的。不严格地说就是，如果<<在同一个表达式中出现了至少两次的话，那么每一个<<都可以为其左操作数使用尽可能多的表达式，而对于它的右操作数则要使用尽可能少的表达式。在我们的例子中，第一个<<运算符的右操作数是"Hello,world"，左操作数是std::cout；第二个<<运算符的右操作数是std::ednl，左操作数是std::cout<<"Hello,world!"。如果用括号来表明运算符和操作数之间的关系，我们就会看到，该输出表达式等价于

```
(std::cout << "Hello,world!") << std::endl
```

每个<<的行为都取决于它的操作数类型。第一个<<的左操作数是std::cout，而std::cout的类型是std::ostream。它的右操作数是一个字符串直接量，我们将在§10.2中开始讨论字符串直接量的类型。根据这些操作数类型，<<会把它的右操作数的字符写到左操作数所指示的流中，它的结果就是它的左操作数。

因此，第二个<<的左操作数就是一个表达式，它会产生类型为std::ostream的std::cout；它的右操作数则是std::endl，std::endl是一个**控制器**（manipulator）。控制器的关键性质是：如果把一个控制器写到流中，那么我们就可以控制这个流了。而为此单单将字符写到流中是不够的，我们还要通过其他的途径来实现对流的控制。如果<<的左操作数的类型是std::ostream，而右操作数是一个控制器，那么<<就会对特定的流执行控制器所指定的动作，同时它会返回流作为其结果。就std::endl而言，我们所做的动作是结束当前的输出行。

因此，整个表达式所产生的值是 std::cout，此外，作为其副作用，它还会把 Hello,world! 写到标准输出流并结束输出行。当我们在表达式后面紧接了一个分号时，就表明我们让系统环境丢掉了这个返回值——这是合理的，因为我们仅仅对副作用感兴趣。

名字的**作用域**是程序的一部分，只有在这一部分中这个名字是有意义的。C++有几种不同的作用域，在这个程序中，我们已经见到了其中的两种。

我们用到的第一种作用域是名字空间，正如我们刚才看到的那样，名字空间是一个相关名称的集合。标准库在一个名为 std 的名字空间中定义了它所提供的所有名称，这样它就可以避免跟我们自己定义的名称发生冲突了——前提是我们还没有愚蠢到想要定义 std 这个名称。在我们使用一个标准库所提供的名称时，我们必须指明所需要的那个名称是来自库的；例如，std::cout 表示 cout 是在名为 std 的名字空间中定义的。

名称 std::cout 是一个**限定名称**，它使用了::运算符。我们把::运算符称为**作用域运算符**。::的左边是一个（很可能是限定的）作用域的名称，就 std::cout 而言，这个名称是一个名为 std 的名字空间。::的右边也是一个名称，这个名称是在左边命名的作用域中定义的。因此，std::cout 表示“在（名字空间）作用域 std 中的名称 cout”。

花括号是另一种作用域。main 的函数体（以及每个函数的函数体）本身就是一个作用域。在这样的一个简单的程序中，我们可能不会对这个事实有太大的兴趣，但它几乎会跟每一个我们所编写的函数都有关联。

0.8 小结

尽管我们编写的程序很简单，但是在这一章中，我们还是涉及了很多的问题。在这里，我们想进一步讨论一下我们所介绍的知识，因此，在继续往下阅读之前，请你先透彻地理解本章的内容。

为了帮你达到这个目的，在这一章（以及在除了第 16 章以外的每一章中）我们都会以一个名为“小结”的章节和一系列的习题来结束我们的讨论。小结概述并偶尔扩展了正文中的信息。读者们有必要去看一下各章的小结，因为它们可以让你们加深对我们在各章中所介绍的概念的理解。

程序结构：C++程序通常具有**自由风格**，也就是说，只是在防止相邻的符号混在一起的时候，我们才必须使用空白符。另外，新行（也就是系统环境从程序的一行转换到下一行时所用的方式）也是另一种形式的空白，此外它并没有其他的特别含义。程序中空白符出现位置的不同可能会提高（或降低）程序的可读性。一般来说，我们应该提高程序的可读性。

以下的三个实体不能具有自由风格：

字符串直接量	用双引号括住的字符；不可以跨行
#include 名称	必须在单独的一行中出现（注释除外）
//注释	//后面可以跟着任何东西；结束于当前行的行尾

以/*开始的注释具有自由风格；它结束于第一个相邻的*/并且可以跨越多行。

类型定义了数据结构以及对这些数据结构的操作。C++有两种形式的类型：核心语言提供的内建类型，如 int；定义在核心语言之外的类型，如 std::cout。

名字空间是一种把相关名称聚集在一起的技术。来自标准库的名称是在名为 std 的名字空间中定义的。

字符串直接量从双引号（"）开始而且结束于双引号；每个字符串直接量都必须全部出现在程序中的一行中。如果字符串直接量中的字符是跟在反斜杠（\）后面的，那么它们会具有特殊的含义：

\n	换行符
\t	水平制表符
\b	回退符
\"	我们把这个符号当作字符串的一部分而不是把它当作字符串的结束符
\'	与字符串直接量中的‘具有一样的意义，用来保持字符直接量的一致性（§1.2）
\\	若字符串中包括一个\，那么就可以把接下来的字符当作普通字符来看待

在§10.2 和§A.2.1.3 中我们会对字符串直接量做进一步的讨论。

定义和头文件：C++程序使用的每一个名称都必须具有一个相应的定义。标准库在头文件中定义它的名称，程序可以通过#include 指令来访问这些名称。名称必须在使用前定义；因此，我们必须在使用头文件中的任何名称之前编写#include 指令。<iostream>头定义了库中的输入/输出工具。

主函数（main）：所有的 C++程序都必须定义且只能定义一个名为 main 的函数，这个函数返回一个 int 类型的值。系统环境通过调用 main 函数来运行程序。main 函数返回值为 0 意味着运行成功；返回值不是 0 则意味着失败。一般来说，函数必须包含至少一条的 return 语句，而且在函数的最后一定要有 return 语句。但 main 比较特殊：它可以没有返回语句；如果是这样的话，编译器就会假设它返回 0。尽管如此，我们还是应该养成在 main 函数中明确声明一条返回语句的好习惯。

花括号和分号：在 C++程序中，这些并不引人注目的符号是很重要的。我们经常会因为它们的微不足道而忽略了它们。它们之所以如此重要是因为，假如我们遗忘了它们中的某一个，那么编译器就很有可能会发出难于理解的诊断信息。

出现在花括号中的零条或多条语句也是一条语句，它要求编译器实现按它们出现的先后次序来顺序执行这些语句。就算函数的函数体只含有一条语句，我们也必须用花括号来括住它。在一对匹配的花括号之间的的语句构成了一个作用域。

如果一个表达式的后面跟着一个分号，那这个表达式也是一条语句。它的执行只是为了获得副作用，实际产生的结果将会被忽略。表达式是可选的：忽略了表达式就会产生一条空语句，这条语句是没有任何作用的。

输出：对 std::cout<<e 进行计算会把 e 的值写到标准输出流中，同时这个计算会产生类型