```
    // 提示用户输入两个数
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;       // 保存我们读入的输入数据的变量
    std::cin >> v1 >> v2;     // 读取输入数据
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

在本书中，我们用楷体来突出显示注释。在实际程序中，注释文本的显示形式是否区别于程序代码文本的显示，依赖于你所使用的程序设计环境是否提供这一特性。

注释界定符不能嵌套

界定符对形式的注释是以/*开始，以*/结束的。因此，一个注释不能嵌套在另一个注释之内。编译器对这类问题所给出的错误信息可能是难以理解、令人迷惑的。例如，在你的系统中编译下面的程序，就会产生错误：

```
/*
 * 注释对/* */不能嵌套。
 * “不能嵌套”几个字会被认为是源码，
 * 像剩余程序一样处理
 */
int main()
{
    return 0;
}
```

我们通常需要在调试期间注释掉一些代码。由于这些代码可能包含界定符对形式的注释，因此可能导致注释嵌套错误，因此最好的方式是用单行注释方式注释掉代码段的每一行。

```
// /*
// * 单行注释中的任何内容都会被忽略
// * 包括嵌套的注释对也一样会被忽略
// */
```

1.3 节练习

11

练习 1.7：编译一个包含不正确的嵌套注释的程序，观察编译器返回的错误信息。

练习 1.8：指出下列哪些输出语句是合法的（如果有的话）：

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /* "*/" /* "/*" */;
```

预测编译这些语句会产生什么样的结果，实际编译这些语句来验证你的答案（编写一个小程序，每次将上述一条语句作为其主体），改正每个编译错误。

1.4 控制流

语句一般是顺序执行的：语句块的第一条语句首先执行，然后是第二条语句，依此类推。当然，少数程序，包括我们解决书店问题的程序，都可以写成只有顺序执行的形式。但程序设计语言提供了多种不同的控制流语句，允许我们写出更为复杂的执行路径。

1.4.1 while 语句

while 语句反复执行一段代码，直至给定条件为假为止。我们可以用 while 语句编写一段程序，求 1 到 10 这 10 个数之和：

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // 只要 val 的值小于等于 10，while 循环就会持续执行
    while (val <= 10) {
        sum += val;  // 将 sum + val 赋予 sum
        ++val;       // 将 val 加 1
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

我们编译并执行这个程序，它会打印出

```
Sum of 1 to 10 inclusive is 55
```

与之前的例子一样，我们首先包含头文件 iostream，然后定义 main。在 main 中我们定义两个 int 变量：sum 用来保存和；val 用来表示从 1 到 10 的每个数。我们将 sum 的初值设置为 0，val 从 1 开始。

12

这个程序的新内容是 while 语句。while 语句的形式为

```
while (condition)
    statement
```

while 语句的执行过程是交替地检测 *condition* 条件和执行关联的语句 *statement*，直至 *condition* 为假时停止。所谓**条件**（condition）就是一个产生真或假的结果的表达式。只要 *condition* 为真，*statement* 就会被执行。当执行完 *statement*，会再次检测 *condition*。如果 *condition* 仍为真，*statement* 再次被执行。while 语句持续地交替检测 *condition* 和执行 *statement*，直至 *condition* 为假为止。

在本程序中，while 语句是这样的

```
// 只要 val 的值小于等于 10，while 循环就会持续执行
while (val <= 10) {
    sum += val;  // 将 sum + val 赋予 sum
    ++val;       // 将 val 加 1
}
```

条件中使用了**小于等于运算符**（**<=**）来比较 val 的当前值和 10。只要 val 小于等于 10，条件即为真。如果条件为真，就执行 while 循环体。在本例中，循环体是由两条语句组

成的语句块：

```
{
    sum += val;      // 将 sum + val 赋予 sum
    ++val;           // 将 val 加 1
}
```

所谓语句块（block），就是用花括号包围的零条或多条语句的序列。语句块也是语句的一种，在任何要求使用语句的地方都可以使用语句块。在本例中，语句块的第一条语句使用了**复合赋值运算符**（+=）。此运算符将其右侧的运算对象加到左侧运算对象上，将结果保存到左侧运算对象中。它本质上与一个加法结合一个**赋值**（assignment）是相同的：

```
sum = sum + val; // 将 sum + val 赋予 sum
```

因此，语句块中第一条语句将 val 的值加到当前和 sum 上，并将结果保存在 sum 中。

下一条语句

```
++val; // 将 val 加 1
```

使用**前缀递增运算符**（++）。递增运算符将运算对象的值增加 1。++val 等价于 val=val+1。

执行完 while 循环体后，循环会再次对条件进行求值。如果 val 的值（现在已经增加了）仍然小于等于 10，则 while 的循环体会再次执行。循环连续检测条件、执行循环体，直至 val 不再小于等于 10 为止。

一旦 val 大于 10，程序跳出 while 循环，继续执行 while 之后的语句。在本例中，继续执行打印输出语句，然后执行 return 语句完成 main 程序。

1.4.1 节练习

13

练习 1.9：编写程序，使用 while 循环将 50 到 100 的整数相加。

练习 1.10：除了++运算符将运算对象的值增加 1 之外，还有一个递减运算符（—）实现将值减少 1。编写程序，使用递减运算符在循环中按递减顺序打印出 10 到 0 之间的整数。

练习 1.11：编写程序，提示用户输入两个整数，打印出这两个整数所指定的范围内的所有整数。

1.4.2 for 语句

在我们的 while 循环例子中，使用了变量 val 来控制循环执行次数。我们在循环条件中检测 val 的值，在 while 循环体中将 val 递增。

这种在循环条件中检测变量、在循环体中递增变量的模式使用非常频繁，以至于 C++ 语言专门定义了第二种循环语句——**for 语句**，来简化符合这种模式的语句。可以用 for 语句来重写从 1 加到 10 的程序：

```
#include <iostream>
int main()
{
    int sum = 0;
    // 从 1 加到 10
    for (int val = 1; val <= 10; ++val)
```

```
        sum += val; // 等价于 sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

与之前一样，我们定义了变量 sum，并将其初始化为 0。在此版本中，val 的定义是 for 语句的一部分：

```
for (int val = 1; val <= 10; ++val)
    sum += val;
```

每个 for 语句都包含两部分：循环头和循环体。循环头控制循环体的执行次数，它由三部分组成：一个初始化语句（*init-statement*）、一个循环条件（*condition*）以及一个表达式（*expression*）。在本例中，初始化语句为

```
int val = 1
```

它定义了一个名为 val 的 int 型对象，并为其赋初值 1。变量 val 仅在 for 循环内部存在，在循环结束之后是不能使用的。初始化语句只在 for 循环入口处执行一次。循环条件

```
val <= 10
```

14 比较 val 的值和 10。循环体每次执行前都会先检查循环条件。只要 val 小于等于 10，就会执行 for 循环体。表达式在 for 循环体之后执行。在本例中，表达式

```
++val
```

使用前缀递增运算符将 val 的值增加 1。执行完表达式后，for 语句重新检测循环条件。如果 val 的新值仍然小于等于 10，就再次执行 for 循环体。执行完循环体后，再次将 val 的值增加 1。循环持续这一过程直至循环条件为假。

在此循环中，for 循环体执行加法

```
sum += val; // 等价于 sum = sum + val
```

简要重述一下 for 循环的总体执行流程：

1. 创建变量 val，将其初始化为 1。
2. 检测 val 是否小于等于 10。若检测成功，执行 for 循环体。若失败，退出循环，继续执行 for 循环体之后的第一条语句。
3. 将 val 的值增加 1。
4. 重复第 2 步中的条件检测，只要条件为真就继续执行剩余步骤。

1.4.2 节练习

练习 1.12：下面的 for 循环完成了什么功能？sum 的终值是多少？

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

练习 1.13：使用 for 循环重做 1.4.1 节中的所有练习（第 11 页）。

练习 1.14：对比 for 循环和 while 循环，两种形式的优缺点各是什么？

练习 1.15：编写程序，包含第 14 页“再探编译”中讨论的常见错误。熟悉编译器生成的错误信息。

1.4.3 读取数量不定的输入数据

在前一节中，我们编写程序实现了 1 到 10 这 10 个整数求和。扩展此程序一个很自然的方向是实现对用户输入的一组数求和。在这种情况下，我们预先不知道要对多少个数求和，这就需要不断读取数据直至没有新的输入为止：

```
#include <iostream>                                          15
int main()
{
    int sum = 0, value = 0;
    // 读取数据直到遇到文件尾，计算所有读入的值的和
    while (std::cin >> value)
        sum += value; // 等价于 sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

如果我们输入

```
3 4 5 6
```

则程序会输出

```
Sum is: 18
```

main 的首行定义了两个名为 sum 和 value 的 int 变量，均初始化为 0。我们使用 value 保存用户输入的每个数，数据读取操作是在 while 的循环条件中完成的：

```
while (std::cin >> value)
```

while 循环条件的求值就是执行表达式

```
std::cin >> value
```

此表达式从标准输入读取下一个数，保存在 value 中。输入运算符（参见 1.2 节，第 7 页）返回其左侧运算对象，在本例中是 std::cin。因此，此循环条件实际上检测的是 std::cin。

当我们使用一个 istream 对象作为条件时，其效果是检测流的状态。如果流是有效的，即流未遇到错误，那么检测成功。当遇到文件结束符（end-of-file），或遇到一个无效输入时（例如读入的值不是一个整数），istream 对象的状态会变为无效。处于无效状态的 istream 对象会使条件变为假。

因此，我们的 while 循环会一直执行直至遇到文件结束符（或输入错误）。while 循环体使用复合赋值运算符将当前值加到 sum 上。一旦条件失败，while 循环将会结束。我们将执行下一条语句，打印 sum 的值和一个 endl。

从键盘输入文件结束符

当从键盘向程序输入数据时，对于如何指出文件结束，不同操作系统有不同的约定。在 Windows 系统中，输入文件结束符的方法是敲 Ctrl+Z（按住 Ctrl 键的同时按 Z 键），然后按 Enter 或 Return 键。在 UNIX 系统中，包括 Mac OS X 系统中，文件结束符输入是用 Ctrl+D。

16

再探编译

编译器的一部分工作是寻找程序文本中的错误。编译器没有能力检查一个程序是否按照其作者的意图工作，但可以检查形式（form）上的错误。下面列出了一些最常见的编译器可以检查出的错误。

语法错误（syntax error）：程序员犯了 C++语言文法上的错误。下面程序展示了一些常见的语法错误；每条注释描述了下一行中语句存在的错误：

```
// 错误：main 的参数列表漏掉了
int main ( {
    // 错误：endl 后使用了冒号而非分号
    std::cout << "Read each file." << std::endl:
    // 错误：字符串字面常量的两侧漏掉了引号
    std::cout << Update master. << std::endl;
    // 错误：漏掉了第二个输出运算符
    std::cout << "Write new master." std::endl;
    // 错误：return 语句漏掉了分号
    return 0
}
```

类型错误（type error）：C++中每个数据项都有其类型。例如，10 的类型是 int（或者更通俗地说，“10 是一个 int 型数据”）。单词"hello"，包括两侧的双引号标记，则是一个字符串字面值常量。一个类型错误的例子是，向一个期望参数为 int 的函数传递了一个字符串字面值常量。

声明错误（declaration error）：C++程序中的每个名字都要先声明后使用。名字声明失败通常会导致一条错误信息。两种常见的声明错误是：对来自标准库的名字忘记使用 std::、标识符名字拼写错误：

```
#include <iostream>
int main()
{
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2; // 错误：使用了"v"而非"v1"
    // 错误：cout 未定义；应该是 std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

错误信息通常包含一个行号和一条简短描述，描述了编译器认为的我们所犯的错误。按照报告的顺序来逐个修正错误，是一种好习惯。因为一个单个错误常常会具有传递效应，导致编译器在其后报告比实际数量多得多的错误信息。另一个好习惯是在每修

正一个错误后就立即重新编译代码，或者最多是修正了一小部分明显的错误后就重新编译。这就是所谓的“编辑-编译-调试”（edit-compile-debug）周期。

1.4.3 节练习

17

练习 1.16：编写程序，从 cin 读取一组数，输出其和。

1.4.4 if 语句

与大多数语言一样，C++也提供了 **if 语句**来支持条件执行。我们可以用 if 语句写一个程序，来统计在输入中每个值连续出现了多少次：

```
#include <iostream>
int main()
{
    // currVal 是我们正在统计的数；我们将读入的新值存入 val
    int currVal = 0, val = 0;
    // 读取第一个数，并确保确实有数据可以处理
    if (std::cin >> currVal) {
        int cnt = 1;                    // 保存我们正在处理的当前值的个数
        while (std::cin >> val) {       // 读取剩余的数
            if (val == currVal)         // 如果值相同
                ++cnt;                  // 将 cnt 加 1
            else {                      // 否则，打印前一个值的个数
                std::cout << currVal << " occurs "
                          << cnt << " times" << std::endl;
                currVal = val;          // 记住新值
                cnt = 1;                // 重置计数器
            }
        } // while 循环在这里结束
        // 记住打印文件中最后一个值的个数
        std::cout << currVal << " occurs "
                  << cnt << " times" << std::endl;
    } // 最外层的 if 语句在这里结束
    return 0;
}
```

如果我们输入如下内容：

```
42 42 42 42 42 55 55 62 100 100 100
```

则输出应该是：

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

有了之前多个程序的基础，你对这个程序中的大部分代码应该比较熟悉了。程序以两个变量 val 和 currVal 的定义开始：currVal 记录我们正在统计出现次数的那个数；val 则保存从输入读取的每个数。与之前的程序相比，新的内容就是两个 if 语句。第一条 if 语句

18
```
if (std::cin >> currVal) {
    // ...
} //最外层的 if 语句在这里结束
```

保证输入不为空。与 while 语句类似，if 也对一个条件进行求值。第一条 if 语句的条件是读取一个数值存入 currVal 中。如果读取成功，则条件为真，我们继续执行条件之后的语句块。该语句块以左花括号开始，以 return 语句之前的右花括号结束。

如果需要统计出现次数的值，我们就定义 cnt，用来统计每个数值连续出现的次数。与上一小节的程序类似，我们用一个 while 循环反复从标准输入读取整数。

while 的循环体是一个语句块，它包含了第二条 if 语句：

```
if (val == currVal)                    // 如果值相同
    ++cnt;                             // 将 cnt 加 1
else {                                 // 否则，打印前一个值的个数
    std::cout << currVal << " occurs "
              << cnt << " times" << std::endl;
    currVal = val;                     // 记住新值
    cnt = 1;                           // 重置计数器
}
```

这条 if 语句中的条件使用了**相等运算符**（==）来检测 val 是否等于 currVal。如果是，我们执行紧跟在条件之后的语句。这条语句将 cnt 增加 1，表明我们再次看到了 currVal。

如果条件为假，即 val 不等于 currVal，则执行 else 之后的语句。这条语句是一个由一条输出语句和两条赋值语句组成的语句块。输出语句打印我们刚刚统计完的值的出现次数。赋值语句将 cnt 重置为 1，将 currVal 重置为刚刚读入的值 val。

C++用=进行赋值，用==作为相等运算符。两个运算符都可以出现在条件中。一个常见的错误是想在条件中使用==（相等判断），却误用了=。

1.4.4 节练习

练习 1.17：如果输入的所有值都是相等的，本节的程序会输出什么？如果没有重复值，输出又会是怎样的？

练习 1.18：编译并运行本节的程序，给它输入全都相等的值。再次运行程序，输入没有重复的值。

练习 1.19：修改你为 1.4.1 节练习 1.10（第 11 页）所编写的程序（打印一个范围内的数），使其能处理用户输入的第一个数比第二个数小的情况。

19 **关键概念：C++程序的缩进和格式**

C++程序很大程度上是格式自由的，也就是说，我们在哪里放置花括号、缩进、注释以及换行符通常不会影响程序的语义。例如，花括号表示 main 函数体的开始，它可以放在 main 的同一行中；也可以像我们所做的那样，放在下一行的起始位置；还可以放在我们喜欢的其他任何位置。唯一的要求是左花括号必须是 main 的形参列表后第一个非空、非注释的字符。

虽然很大程度上可以按照自己的意愿自由地设定程序的格式，但我们所做的选择会影响程序的可读性。例如，我们可以将整个 main 函数写在很长的单行内，虽然这样是合乎语法的，但会非常难读。

关于 C/C++的正确格式的辩论是无休止的。我们的信条是，不存在唯一正确的风格，但保持一致性是非常重要的。例如，大多数程序员都对程序的组成部分设置恰当的缩进，就像我们在之前的例子中对 main 函数中的语句和循环体所做的那样。对于作为函数界定符的花括号，我们习惯将其放在单独一行中。我们还习惯对复合 IO 表达式设置缩进，以使输入输出运算符排列整齐。其他一些缩进约定也都会令越来越复杂的程序更加清晰易读。

我们要牢记一件重要的事情：其他可能的程序格式总是存在的。当你要选择一种格式风格时，思考一下它会对程序的可读性和易理解性有什么影响，而一旦选择了一种风格，就要坚持使用。

1.5 类简介

在解决书店程序之前，我们还需要了解的唯一一个 C++特性，就是如何定义一个数据结构（data structure）来表示销售数据。在 C++中，我们通过定义一个**类**（class）来定义自己的数据结构。一个类定义了一个类型，以及与其关联的一组操作。类机制是 C++最重要的特性之一。实际上，C++最初的一个设计焦点就是能定义使用上像内置类型一样自然的**类类型**（class type）。

在本节中，我们将介绍一个在编写书店程序中会用到的简单的类。当我们在后续章节中学习了更多关于类型、表达式、语句和函数的知识后，会真正实现这个类。

为了使用类，我们需要了解三件事情：

- 类名是什么？
- 它是在哪里定义的？
- 它支持什么操作？

对于书店程序来说，我们假定类名为 Sales_item，头文件 Sales_item.h 中已经定义了这个类。

如前所见，为了使用标准库设施，我们必须包含相关的头文件。类似的，我们也需要使用头文件来访问为自己的应用程序所定义的类。习惯上，头文件根据其中定义的类的名字来命名。我们通常使用.h 作为头文件的后缀，但也有一些程序员习惯.H、.hpp 或.hxx。标准库头文件通常不带后缀。编译器一般不关心头文件名的形式，但有的 IDE 对此有特定要求。 20

1.5.1 Sales_item 类

Sales_item 类的作用是表示一本书的总销售额、售出册数和平均售价。我们现在不关心这些数据如何存储、如何计算。为了使用一个类，我们不必关心它是如何实现的，只需知道类对象可以执行什么操作。

每个类实际上都定义了一个新的类型，其类型名就是类名。因此，我们的 Sales_item 类定义了一个名为 Sales_item 的类型。与内置类型一样，我们可以定义类类型的变量。当我们写下如下语句

```
Sales_item item;
```

是想表达 item 是一个 Sales_item 类型的对象。我们通常将“一个 Sales_item 类型的对象”简单说成“一个 Sales_item 对象”，或更简单的“一个 Sales_item”。

除了可以定义 Sales_item 类型的变量之外，我们还可以：

- 调用一个名为 isbn 的函数从一个 Sales_item 对象中提取 ISBN 书号。
- 用输入运算符（>>）和输出运算符（<<）读、写 Sales_item 类型的对象。
- 用赋值运算符（=）将一个 Sales_item 对象的值赋予另一个 Sales_item 对象。
- 用加法运算符(+)将两个 Sales_item 对象相加。两个对象必须表示同一本书(相同的 ISBN)。加法结果是一个新的 Sales_item 对象，其 ISBN 与两个运算对象相同，而其总销售额和售出册数则是两个运算对象的对应值之和。
- 使用复合赋值运算符（+=）将一个 Sales_item 对象加到另一个对象上。

关键概念：类定义了行为

当你读这些程序时，一件要牢记的重要事情是，类 Sales_item 的作者定义了类对象可以执行的所有动作。即，Sales_item 类定义了创建一个 Sales_item 对象时会发生什么事情，以及对 Sales_item 对象进行赋值、加法或输入输出运算时会发生什么事情。

一般而言，类的作者决定了类类型对象上可以使用的所有操作。当前，我们所知道的可以在 Sales_item 对象上执行的全部操作就是本节所列出的那些操作。

21 读写 Sales_item

既然已经知道可以对 Sales_item 对象执行哪些操作，我们现在就可以编写使用类的程序了。例如，下面的程序从标准输入读入数据，存入一个 Sales_item 对象中，然后将 Sales_item 的内容写回到标准输出：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // 读入 ISBN 号、售出的册数以及销售价格
    std::cin >> book;
    // 写入 ISBN、售出的册数、总销售额和平均价格
    std::cout << book << std::endl;
    return 0;
}
```

如果输入：

```
0-201-70353-X 4 24.99
```

则输出为：

```
0-201-70353-X 4 99.96 24.99
```

输入表示我们以每本 24.99 美元的价格售出了 4 册书，而输出告诉我们总售出册数为 4，总销售额为 99.96 美元，而每册书的平均销售价格为 24.99 美元。

此程序以两个#include 指令开始，其中一个使用了新的形式。包含来自标准库的头

文件时，也应该用尖括号（< >）包围头文件名。对于不属于标准库的头文件，则用双引号（""）包围。

在 main 中我们定义了一个名为 book 的对象，用来保存从标准输入读取出的数据。下一条语句读取数据存入对象中，第三条语句将对象打印到标准输出上并打印一个 endl。

Sales_item 对象的加法

下面是一个更有意思的例子，将两个 Sales_item 对象相加：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;                  // 读取一对交易记录
    std::cout << item1 + item2 << std::endl;     // 打印它们的和
    return 0;
}
```

如果输入如下内容：

22

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

则输出为：

```
0-201-78345-X 5 110 22
```

此程序开始包含了 Sales_item 和 iostream 两个头文件。然后定义了两个 Sales_item 对象来保存销售记录。我们从标准输入读取数据，存入两个对象之中。输出表达式完成加法运算并打印结果。

值得注意的是，此程序看起来与第 5 页的程序非常相似：读取两个输入数据并输出它们的和。造成如此相似的原因是，我们只不过将运算对象从两个整数变为两个 Sales_item 而已，但读取与打印和的运算方式没有发生任何变化。两个程序的另一个不同之处是，“和”的概念是完全不一样的。对于 int，我们计算传统意义上的和——两个数值的算术加法结果。对于 Sales_item 对象，我们用了一个全新的“和”的概念——两个 Sales_item 对象的成员对应相加的结果。

使用文件重定向

当你测试程序时，反复从键盘敲入这些销售记录作为程序的输入，是非常乏味的。大多数操作系统支持文件重定向，这种机制允许我们将标准输入和标准输出与命名文件关联起来：

```
$ addItems <infile >outfile
```

假定$是操作系统提示符，我们的加法程序已经编译为名为 addItems.exe 的可执行文件（在 UNIX 中是 addItems），则上述命令会从一个名为 infile 的文件读取销售记录，并将输出结果写入到一个名为 outfile 的文件中，两个文件都位于当前目录中。

1.5.1 节练习

练习 1.20：在网站 http://www.informit.com/title/0321714113 上，第 1 章的代码目录中包含了头文件 Sales_item.h。将它拷贝到你自己的工作目录中。用它编写一个程序，读取一组书籍销售记录，将每条记录打印到标准输出上。

练习 1.21：编写程序，读取两个 ISBN 相同的 Sales_item 对象，输出它们的和。

练习 1.22：编写程序，读取多个具有相同 ISBN 的销售记录，输出所有记录的和。

23

1.5.2 初识成员函数

将两个 Sales_item 对象相加的程序首先应该检查两个对象是否具有相同的 ISBN。方法如下：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // 首先检查 item1 和 item2 是否表示相同的书
    if (item1.isbn() == item2.isbn()) {
        std::cout << item1 + item2 << std::endl;
        return 0;    // 表示成功
    } else {
    std::cerr << "Data must refer to same ISBN"
              << std::endl;
    return -1;       // 表示失败
    }
}
```

此程序与上一版本的差别是 if 语句及其 else 分支。即使不了解这个 if 语句的检测条件，我们也很容易理解这个程序在干什么。如果条件成立，如上一版本一样，程序打印计算结果，并返回 0，表明成功。如果条件失败，我们执行跟在 else 之后的语句块，打印一条错误信息，并返回一个错误标识。

什么是成员函数？

这个 if 语句的检测条件

```
item1.isbn() == item2.isbn()
```

调用名为 isbn 的**成员函数**（member function）。成员函数是定义为类的一部分的函数，有时也被称为**方法**（method）。

我们通常以一个类对象的名义来调用成员函数。例如，上面相等表达式左侧运算对象的第一部分

```
item1.isbn()
```

使用**点运算符**（.）来表达我们需要“名为 item1 的对象的 isbn 成员”。点运算符只能用于类类型的对象。其左侧运算对象必须是一个类类型的对象，右侧运算对象必须是该类型的一个成员名，运算结果为右侧运算对象指定的成员。

当用点运算符访问一个成员函数时，通常我们是想（效果也确实是）调用该函数。我们使用**调用运算符**（()）来调用一个函数。调用运算符是一对圆括号，里面放置实参（argument）列表（可能为空）。成员函数 isbn 并不接受参数。因此

```
item1.isbn()
```

调用名为 item1 的对象的成员函数 isbn，此函数返回 item1 中保存的 ISBN 书号。 24

在这个 if 条件中，相等运算符的右侧运算对象也是这样执行的——它返回保存在 item2 中的 ISBN 书号。如果 ISBN 相同，条件为真，否则为假。

1.5.2 节练习

练习 1.23：编写程序，读取多条销售记录，并统计每个 ISBN（每本书）有几条销售记录。

练习 1.24：输入表示多个 ISBN 的多条销售记录来测试上一个程序，每个 ISBN 的记录应该聚在一起。

1.6 书店程序

现在我们已经准备好完成书店程序了。我们需要从一个文件中读取销售记录，生成每本书的销售报告，显示售出册数、总销售额和平均售价。我们假定每个 ISBN 书号的所有销售记录在文件中是聚在一起保存的。

我们的程序会将每个 ISBN 的所有数据合并起来，存入名为 total 的变量中。我们使用另一个名为 trans 的变量保存读取的每条销售记录。如果 trans 和 total 指向相同的 ISBN，我们会更新 total 的值。否则，我们会打印 total 的值，并将其重置为刚刚读取的数据（trans）：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item total; // 保存下一条交易记录的变量
    // 读入第一条交易记录，并确保有数据可以处理
    if (std::cin >> total) {
        Sales_item trans;          // 保存和的变量
        // 读入并处理剩余交易记录
        while (std::cin >> trans) {
            // 如果我们仍在处理相同的书
            if (total.isbn() == trans.isbn())
                total += trans; // 更新总销售额
            else {
                // 打印前一本书的结果
                std::cout << total << std::endl;
                total = trans;  // total 现在表示下一本书的销售额
            }
        }
        std::cout << total << std::endl; // 打印最后一本书的结果
    } else {
```

25

```
            //没有输入! 警告读者
            std::cerr << "No data?!" << std::endl;
            return -1; // 表示失败
        }
        return 0;
    }
```

这是到目前为止我们看到的最复杂的程序了，但它所使用的都是我们已经见过的语言特性。

与往常一样，首先包含要使用的头文件：来自标准库的 iostream 和自己定义的 Sales_item.h。在 main 中，我们定义了一个名为 total 的变量，用来保存一个给定的 ISBN 的数据之和。我们首先读取第一条销售记录，存入 total 中，并检测这次读取操作是否成功。如果读取失败，则意味着没有任何销售记录，于是直接跳到最外层的 else 分支，打印一条警告信息，告诉用户没有输入。

假定已经成功读取了一条销售记录，我们继续执行最外层 if 之后的语句块。这个语句块首先定义一个名为 trans 的对象，它保存读取的销售记录。接下来的 while 语句将读取剩下的所有销售记录。与我们之前的程序一样，while 条件是一个从标准输入读取值的操作。在本例中，我们读取一个 Sales_item 对象，存入 trans 中。只要读取成功，就执行 while 循环体。

while 的循环体是一个单个的 if 语句，它检查 ISBN 是否相等。如果相等，使用复合赋值运算符将 trans 加到 total 中。如果 ISBN 不等，我们打印保存在 total 中的值，并将其重置为 trans 的值。在执行完 if 语句后，返回到 while 的循环条件，读取下一条销售记录，如此反复，直至所有销售记录都处理完。

当 while 语句终止时，total 保存着文件中最后一个 ISBN 的数据。我们在语句块的最后一条语句中打印这最后一个 ISBN 的 total 值，至此最外层 if 语句就结束了。

1.6 节练习

练习 1.25：借助网站上的 Sales_item.h 头文件，编译并运行本节给出的书店程序。

小结

本章介绍了足够多的C++语言的知识，以使你能够编译、运行简单的C++程序。我们看到了如何定义一个main函数，它是操作系统执行你的程序的调用入口。我们还看到了如何定义变量，如何进行输入输出，以及如何编写if、for和while语句。本章最后介绍了C++中最基本的特性——类。在本章中，我们看到了，对于其他人定义的一个类，我们应该如何创建、使用其对象。在后续章节中，我们将介绍如何定义自己的类。

术语表

参数（实参，argument）向函数传递的值。

赋值（assignment）抹去一个对象的当前值，用一个新值取代之。

程序块（block）零条或多条语句的序列，用花括号包围。

缓冲区（buffer）一个存储区域，用于保存数据。IO设施通常将输入（或输出）数据保存在一个缓冲区中，读写缓冲区的动作与程序中的动作是无关的。我们可以显式地刷新输出缓冲，以便强制将缓冲区中的数据写入输出设备。默认情况下，读cin会刷新cout；程序非正常终止时也会刷新cout。

内置类型（built-in type）由语言定义的类型，如int。

Cerr 一个ostream对象，关联到标准错误，通常写入到与标准输出相同的设备。默认情况下，写到cerr的数据是不缓冲的。cerr通常用于输出错误信息或其他不属于程序正常逻辑的输出内容。

字符串字面值常量（character string literal）术语string literal的另一种叫法。

cin 一个istream对象，用来从标准输入读取数据。

类（class）一种用于定义自己的数据结构及其相关操作的机制。类是C++中最基本的特性之一。标准库类型中，如istream和ostream都是类。

类类型（class type）类定义的类型。类名即为类型名。

clog 一个ostream对象，关联到标准错误。默认情况下，写到clog的数据是被缓冲的。clog通常用于报告程序的执行信息，存入一个日志文件中。

注释（comment）被编译器忽略的程序文本。C++有两种类型的注释：单行注释和界定符对注释。单行注释以//开始，从//到行尾的所有内容都是注释。界定符对注释以/*开始，其后的所有内容都是注释，直至遇到*/为止。

条件（condition）求值结果为真或假的表达式。通常用值0表示假，用非零值表示真。

cout 一个ostream对象，用于将数据写入标准输出。通常用于程序的正常输出内容。

花括号（curly brace）花括号用于划定程序块边界。左花括号（{）为程序块开始，右花括号（}）为结束。

数据结构（data structure）数据及其上所允许的操作的一种逻辑组合。

编辑-编译-调试（edit-compile-debug）使程序能正确执行的开发过程。

文件结束符（end-of-file） 系统特定的标识，指出文件中无更多数据了。

表达式（expression）最小的计算单元。一个表达式包含一个或多个运算对象，通常还包含一个或多个运算符。表达式求值会产生一个结果。例如，假设i和j是int对象，则i+j是一个表达式，它产生两个

int 值的和。

for 语句（for statement）迭代语句，提供重复执行能力。通常用来将一个计算反复执行指定次数。

函数（function）具名的计算单元。

函数体（function body）语句块，定义了函数所执行的动作。

函数名（function name） 函数为人所知的名字，也用来进行函数调用。

头文件（header）使类或其他名字的定义可被多个程序使用的一种机制。程序通过 #include 指令使用头文件。

if 语句（if statement）根据一个特定条件的值进行条件执行的语句。如果条件为真，执行 if 语句体。否则，执行 else 语句体（如果存在的话）。

初始化（initialize）在一个对象创建的时候赋予它一个值。

iostream 头文件，提供了面向流的输入输出的标准库类型。

istream 提供了面向流的输入的库类型。

库类型（library type）标准库定义的类型，如 istream。

main 操作系统执行一个 C++程序时所调用的函数。每个程序必须有且只有一个命名为 main 的函数。

操纵符（manipulator）对象，如 std::endl，在读写流的时候用来“操纵”流本身。

成员函数（member function）类定义的操作。通常通过调用成员函数来操作特定对象。

方法（method）成员函数的同义术语。

命名空间（namespace）将库定义的名字放在一个单一位置的机制。命名空间可以帮助避免不经意的名字冲突。C++标准库定义的名字在命名空间 std 中。

ostream 标准库类型，提供面向流的输出。

形参列表（parameter list）函数定义的一部分，指出调用函数时可以使用什么样的实参，可能为空列表。

返回类型（return type）函数返回值的类型。

源文件（source file）包含 C++程序的文件。

标准错误（standard error）输出流，用于报告错误。标准输出和标准错误通常关联到程序执行所在的窗口。

标准输入（standard input）输入流，通常与程序执行所在窗口相关联。

标准库（standard library）一个类型和函数的集合，每个 C++编译器都必须支持。标准库提供了支持 IO 操作的类型。C++程序员倾向于用“库”指代整个标准库，还倾向于用库类型表示标准库的特定部分，例如用“iostream 库”表示标准库中定义 IO 类的部分。

标准输出（standard output）输出流，通常与程序执行所在窗口相关联。

语句（statement）程序的一部分，指定了当程序执行时进行什么动作。一个表达式接一个分号就是一条语句；其他类型的语句包括语句块、if 语句、for 语句和 while 语句，所有这些语句内都包含其他语句。

std 标准库所使用的命名空间。std::cout 表示我们要使用定义在命名空间 std 中的名字 cout。

字符串常量（string literal）零或多个字符组成的序列，用双引号包围（"a string literal"）。

未初始化的变量（uninitialized variable）未赋予初值的变量。类类型的变量如果未指定初值，则按类定义指定的方式进行初始化。定义在函数内部的内置类型变量默认是不初始化的，除非有显式的初始化语句。试图使用一个未初始化变量的值是错误的。未初始化变量是 bug 的常见成因。

变量（variable）具名对象。

while 语句（while statement）迭代语句，提供重复执行直至一个特定条件为假的机制。循环体会执行零次或多次，依赖于循环条件求值结果。

()运算符（() operator）调用运算符。跟随在函数名之后的一对括号“()”，起到调用函数的效果。传递给函数的实参放置在括号内。

++运算符（++ operator）递增运算符。将运算对象的值加 1，`++i` 等价于 `i=i+1`。

+=运算符（+= operator）复合赋值运算符，将右侧运算对象加到左侧运算对象上；`a+=b` 等价于 `a=a+b`。

.运算符（. operator）点运算符。左侧运算对象必须是一个类类型对象，右侧运算对象必须是此对象的一个成员的名字。运算结果即为该对象的这个成员。

::运算符（:: operator）作用域运算符。其用处之一是访问命名空间中的名字。例如，`std::cout` 表示命名空间 `std` 中的名字 `cout`。

=运算符（= operator）将右侧运算对象的值赋予左侧运算对象所表示的对象。

—运算符（— operator）递减运算符。将运算对象的值减 1，`--i` 等价于 `i=i-1`。

<<运算符（<< operator）输出运算符。将右侧运算对象的值写到左侧运算对象表示的输出流：`cout << "hi"`表示将 `hi` 写到标准输出。输出运算符可以连接：`cout << "hi" << "bye"`表示将输出 `hibye`。

>>运算符（>> operator）输入运算符。从左侧运算对象所指定的输入流读取数据，存入右侧运算对象中：`cin >> i` 表示从标准输入读取下一个值，存入 `i` 中。输入运算符可以连接：`cin >> i >> j` 表示先读取一个值存入 `i`，再读取一个值存入 `j`。

#include 头文件包含指令，使头文件中代码可被程序使用。

==运算符（== operator）相等运算符。检测左侧运算对象是否等于右侧运算对象。

!=运算符（!= operator）不等运算符。检测左侧运算对象是否不等于右侧运算对象。

<=运算符（<= operator）小于等于运算符。检测左侧运算对象是否小于等于右侧运算对象。

<运算符（< operator）小于运算符。检测左侧运算对象是否小于右侧运算对象。

>=运算符（>= operator）大于等于运算符。检测左侧运算对象是否大于等于右侧运算对象。

>运算符（> operator）大于运算符。检测左侧运算对象是否大于右侧运算对象。

第 I 部分
C++基础

内容

任何常用的编程语言都具备一组公共的语法特征，不同语言仅在特征的细节上有所区别。要想学习并掌握一种编程语言，理解其语法特征的实现细节是第一步。最基本的特征包括：

- 整型、字符型等内置类型
- 变量，用来为对象命名
- 表达式和语句，用于操纵上述数据类型的具体值
- `if` 或 `while` 等控制结构，这些结构允许我们有选择地执行一些语句或者重复地执行一些语句
- 函数，用于定义可供随时调用的计算单元

大多数编程语言通过两种方式来进一步补充其基本特征：一是赋予程序员自定义数据类型的权利，从而实现对语言的扩展；二是将一些有用的功能封装成库函数提供给程序员。

30 与大多数编程语言一样，C++的对象类型决定了能对该对象进行的操作，一条表达式是否合法依赖于其中参与运算的对象的类型。一些语言，如 Smalltalk 和 Python 等，在程序运行时检查数据类型；与之相反，C++是一种静态数据类型语言，它的类型检查发生在编译时。因此，编译器必须知道程序中每一个变量对应的数据类型。

C++提供了一组内置数据类型、相应的运算符以及为数不多的几种程序流控制语句，这些元素共同构成了 C++语言的基本形态。以这些元素为基础，我们可以编写出规模庞大、结构复杂、用于解决实际问题的软件系统。仅就 C++的基本形态来说，它是一种简单的编程语言，其强大的能力显示于它对程序员自定义数据结构的支持。这种支持作用巨大，显而易见的一个事实是，C++语言的缔造者无须洞悉所有程序员的要求，而程序员恰好可以通过自主定义新的数据结构来使语言满足他们各自的需求。

C++中最重要的语法特征应该就是类了，通过它，程序员可以定义自己的数据类型。为了与 C++的内置类型区别开来，它们通常被称为“类类型（class type）”。在一些编程语言中，程序员自定义的新类型仅能包含数据成员；另外一些语言，比如 C++，则允许新类型中既包含数据成员，也包含函数成员。C++语言主要的一个设计目标就是让程序员自定义的数据类型像内置类型一样好用。基于此，标准 C++库实现了丰富的类和函数。

本书第Ⅰ部分的主题是学习 C++语言的基础知识，这也是掌握 C++语言的第一步。第 2 章详述内置类型，并初步介绍了自定义数据类型的方法。第 3 章介绍了两种最基本的数据类型：字符串和向量。C++和许多编程语言所共有的一种底层数据结构——数组也在本章有所提及。接下来，第 4～6 章依次介绍了表达式、语句和函数。作为第Ⅰ部分的最后一章，第 7 章描述了如何构建我们自己的类，完成这一任务需要综合运用之前各章所介绍的知识。

第 2 章
变量和基本类型

内容

数据类型是程序的基础：它告诉我们数据的意义以及我们能在数据上执行的操作。

C++语言支持广泛的数据类型。它定义了几种基本内置类型（如字符、整型、浮点数等），同时也为程序员提供了自定义数据类型的机制。基于此，C++标准库定义了一些更加复杂的数据类型，比如可变长字符串和向量等。本章将主要讲述内置类型，并带领大家初步了解 C++语言是如何支持更复杂数据类型的。

32 数据类型决定了程序中数据和操作的意义。如下所示的语句是一个简单示例：

```
i = i + j;
```

其含义依赖于 i 和 j 的数据类型。如果 i 和 j 都是整型数，那么这条语句执行的就是最普通的加法运算。然而，如果 i 和 j 是 Sales_item 类型的数据（参见 1.5.1 节，第 17 页），则上述语句把这两个对象的成分相加。

2.1 基本内置类型

C++定义了一套包括**算术类型**（arithmetic type）和**空类型**（void）在内的基本数据类型。其中算术类型包含了字符、整型数、布尔值和浮点数。空类型不对应具体的值，仅用于一些特殊的场合，例如最常见的是，当函数不返回任何值时使用空类型作为返回类型。

2.1.1 算术类型

算术类型分为两类：**整型**（integral type，包括字符和布尔类型在内）和浮点型。

算术类型的尺寸（也就是该类型数据所占的比特数）在不同机器上有所差别。表 2.1 列出了 C++标准规定的尺寸的最小值，同时允许编译器赋予这些类型更大的尺寸。某一类型所占的比特数不同，它所能表示的数据范围也不一样。

表 2.1：C++：算术类型

类型	含义	最小尺寸
bool	布尔类型	未定义
char	字符	8 位
wchar_t	宽字符	16 位
char16_t	Unicode 字符	16 位
char32_t	Unicode 字符	32 位
short	短整型	16 位
int	整型	16 位
long	长整型	32 位
long long	长整型	64 位
float	单精度浮点数	6 位有效数字
double	双精度浮点数	10 位有效数字
long double	扩展精度浮点数	10 位有效数字

布尔类型（bool）的取值是真（true）或者假（false）。

C++提供了几种字符类型，其中多数支持国际化。基本的字符类型是 char，一个 char 的空间应确保可以存放机器基本字符集中任意字符对应的数字值。也就是说，一个 char 的大小和一个机器字节一样。

33 其他字符类型用于扩展字符集，如 wchar_t、char16_t、char32_t。wchar_t 类型用于确保可以存放机器最大扩展字符集中的任意一个字符，类型 char16_t 和 char32_t 则为 Unicode 字符集服务（Unicode 是用于表示所有自然语言中字符的标准）。

除字符和布尔类型之外，其他整型用于表示（可能）不同尺寸的整数。C++语言规定一个 int 至少和一个 short 一样大，一个 long 至少和一个 int 一样大，一个 long long

至少和一个 long 一样大。其中，数据类型 long long 是在 C++11 中新定义的。 C++ 11

内置类型的机器实现

计算机以比特序列存储数据，每个比特非 0 即 1，例如：

```
00011011011100010110010000111011 ...
```

大多数计算机以 2 的整数次幂个比特作为块来处理内存，可寻址的最小内存块称为“字节（byte）”，存储的基本单元称为“字（word）”，它通常由几个字节组成。在 C++语言中，一个字节要至少能容纳机器基本字符集中的字符。大多数机器的字节由 8 比特构成，字则由 32 或 64 比特构成，也就是 4 或 8 字节。

大多数计算机将内存中的每个字节与一个数字（被称为“地址（address）”）关联起来，在一个字节为 8 比特、字为 32 比特的机器上，我们可能看到一个字的内存区域如下所示：

736424	0	0	1	1	1	0	1	1
736425	0	0	0	1	1	0	1	1
736426	0	1	1	1	0	0	0	1
736427	0	1	1	0	0	1	0	0

其中，左侧是字节的地址，右侧是字节中 8 比特的具体内容。

我们能够使用某个地址来表示从这个地址开始的大小不同的比特串，例如，我们可能会说地址 736424 的那个字或者地址 736427 的那个字节。为了赋予内存中某个地址明确的含义，必须首先知道存储在该地址的数据的类型。类型决定了数据所占的比特数以及该如何解释这些比特的内容。

如果位置 736424 处的对象类型是 float，并且该机器中 float 以 32 比特存储，那么我们就能知道这个对象的内容占满了整个字。这个 float 数的实际值依赖于该机器是如何存储浮点数的。或者如果位置 736424 处的对象类型是 unsigned char，并且该机器使用 ISO-Latin-1 字符集，则该位置处的字节表示一个分号。

浮点型可表示单精度、双精度和扩展精度值。C++标准指定了一个浮点数有效位数的最小值，然而大多数编译器都实现了更高的精度。通常，float 以 1 个字（32 比特）来表示，double 以 2 个字（64 比特）来表示，long double 以 3 或 4 个字（96 或 128 比特）来表示。一般来说，类型 float 和 double 分别有 7 和 16 个有效位；类型 long double 则常常被用于有特殊浮点需求的硬件，它的具体实现不同，精度也各不相同。 34

带符号类型和无符号类型

除去布尔型和扩展的字符型之外，其他整型可以划分为**带符号的**（signed）和**无符号的**（unsigned）两种。带符号类型可以表示正数、负数或 0，无符号类型则仅能表示大于等于 0 的值。

类型 int、short、long 和 long long 都是带符号的，通过在这些类型名前添加 unsigned 就可以得到无符号类型，例如 unsigned long。类型 unsigned int 可以缩写为 unsigned。

与其他整型不同，字符型被分为了三种：char、signed char 和 unsigned char。

特别需要注意的是：类型 char 和类型 signed char 并不一样。尽管字符型有三种，但是字符的表现形式却只有两种：带符号的和无符号的。类型 char 实际上会表现为上述两种形式中的一种，具体是哪种由编译器决定。

无符号类型中所有比特都用来存储值，例如，8 比特的 unsigned char 可以表示 0 至 255 区间内的值。

C++标准并没有规定带符号类型应如何表示，但是约定了在表示范围内正值和负值的量应该平衡。因此，8 比特的 signed char 理论上应该可以表示-127 至 127 区间内的值，大多数现代计算机将实际的表示范围定为-128 至 127。

建议：如何选择类型

和 C 语言一样，C++的设计准则之一也是尽可能地接近硬件。C++的算术类型必须满足各种硬件特质，所以它们常常显得繁杂而令人不知所措。事实上，大多数程序员能够（也应该）对数据类型的使用做出限定从而简化选择的过程。以下是选择类型的一些经验准则：

- 当明确知晓数值不可能为负时，选用无符号类型。
- 使用 int 执行整数运算。在实际应用中，short 常常显得太小而 long 一般和 int 有一样的尺寸。如果你的数值超过了 int 的表示范围，选用 long long。
- 在算术表达式中不要使用 char 或 bool，只有在存放字符或布尔值时才使用它们。因为类型 char 在一些机器上是有符号的，而在另一些机器上又是无符号的，所以如果使用 char 进行运算特别容易出问题。如果你需要使用一个不大的整数，那么明确指定它的类型是 signed char 或者 unsigned char。
- 执行浮点数运算选用 double，这是因为 float 通常精度不够而且双精度浮点数和单精度浮点数的计算代价相差无几。事实上，对于某些机器来说，双精度运算甚至比单精度还快。long double 提供的精度在一般情况下是没有必要的，况且它带来的运行时消耗也不容忽视。

35

2.1.1 节练习

练习 2.1：类型 int、long、long long 和 short 的区别是什么？无符号类型和带符号类型的区别是什么？float 和 double 的区别是什么？

练习 2.2：计算按揭贷款时，对于利率、本金和付款分别应选择何种数据类型？说明你的理由。

2.1.2 类型转换

对象的类型定义了对象能包含的数据和能参与的运算，其中一种运算被大多数类型支持，就是将对象从一种给定的类型**转换**（convert）为另一种相关类型。

当在程序的某处我们使用了一种类型而其实对象应该取另一种类型时，程序会自动进行类型转换，在 4.11 节（第 141 页）中我们将对类型转换做更详细的介绍。此处，有必要说明当给某种类型的对象强行赋了另一种类型的值时，到底会发生什么。

当我们像下面这样把一种算术类型的值赋给另外一种类型时：

```
bool b = 42;                    // b 为真
```

```
int i = b;                  // i 的值为 1
i = 3.14;                   // i 的值为 3
double pi = i;              // pi 的值为 3.0
unsigned char c = -1;       // 假设 char 占 8 比特，c 的值为 255
signed char c2 = 256;       // 假设 char 占 8 比特，c2 的值是未定义的
```

类型所能表示的值的范围决定了转换的过程：

- 当我们把一个非布尔类型的算术值赋给布尔类型时，初始值为 0 则结果为 false，否则结果为 true。
- 当我们把一个布尔值赋给非布尔类型时，初始值为 false 则结果为 0，初始值为 true 则结果为 1。
- 当我们把一个浮点数赋给整数类型时，进行了近似处理。结果值将仅保留浮点数中小数点之前的部分。
- 当我们把一个整数值赋给浮点类型时，小数部分记为 0。如果该整数所占的空间超过了浮点类型的容量，精度可能有损失。
- 当我们赋给无符号类型一个超出它表示范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数。例如，8 比特大小的 unsigned char 可以表示 0 至 255 区间内的值，如果我们赋了一个区间以外的值，则实际的结果是该值对 256 取模后所得的余数。因此，把-1 赋给 8 比特大小的 unsigned char 所得的结果是 255。
- 当我们赋给带符号类型一个超出它表示范围的值时，结果是**未定义的**（undefined）。此时，程序可能继续工作、可能崩溃，也可能生成垃圾数据。

建议：避免无法预知和依赖于实现环境的行为 36

无法预知的行为源于编译器无须（有时是不能）检测的错误。即使代码编译通过了，如果程序执行了一条未定义的表达式，仍有可能产生错误。

不幸的是，在某些情况和/或某些编译器下，含有无法预知行为的程序也能正确执行。但是我们却无法保证同样一个程序在别的编译器下能正常工作，甚至已经编译通过的代码再次执行也可能会出错。此外，也不能认为这样的程序对一组输入有效，对另一组输入就一定有效。

程序也应该尽量避免依赖于实现环境的行为。如果我们把 int 的尺寸看成是一个确定不变的已知值，那么这样的程序就称作**不可移植的**（nonportable）。当程序移植到别的机器上后，依赖于实现环境的程序就可能发生错误。要从过去的代码中定位这类错误可不是一件轻松愉快的工作。

当在程序的某处使用了一种算术类型的值而其实所需的是另一种类型的值时，编译器同样会执行上述的类型转换。例如，如果我们使用了一个非布尔值作为条件（参见 1.4.1 节，第 10 页），那么它会被自动地转换成布尔值，这一做法和把非布尔值赋给布尔变量时的操作完全一样：

```
int i = 42;
if (i)                      // if 条件的值将为 true
   i = 0;
```

如果 i 的值为 0，则条件的值为 false；i 的所有其他取值（非 0）都将使条件为 true。

以此类推，如果我们把一个布尔值用在算术表达式里，则它的取值非 0 即 1，所以一般不宜在算术表达式里使用布尔值。

含有无符号类型的表达式

尽管我们不会故意给无符号对象赋一个负值，却可能（特别容易）写出这么做的代码。例如，当一个算术表达式中既有无符号数又有 int 值时，那个 int 值就会转换成无符号数。把 int 转换成无符号数的过程和把 int 直接赋给无符号变量一样：

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // 输出-84
std::cout << u + i << std::endl; // 如果 int 占 32 位，输出 4294967264
```

在第一个输出表达式里，两个（负）整数相加并得到了期望的结果。在第二个输出表达式里，相加前首先把整数-42 转换成无符号数。把负数转换成无符号数类似于直接给无符号数赋一个负值，结果等于这个负数加上无符号数的模。

当从无符号数中减去一个值时，不管这个值是不是无符号数，我们都必须确保结果不能是一个负值：

37

```
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; // 正确：输出 32
std::cout << u2 - u1 << std::endl; // 正确：不过，结果是取模后的值
```

无符号数不会小于 0 这一事实同样关系到循环的写法。例如，在 1.4.1 节的练习（第 11 页）中需要写一个循环，通过控制变量递减的方式把从 10 到 0 的数字降序输出。这个循环可能类似于下面的形式：

```
for (int i = 10; i >= 0; --i)
    std::cout << i << std::endl;
```

可能你会觉得反正也不打算输出负数，可以用无符号数来重写这个循环。然而，这个不经意的改变却意味着死循环：

```
// 错误：变量 u 永远也不会小于 0，循环条件一直成立
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

来看看当 u 等于 0 时发生了什么，这次迭代输出 0，然后继续执行 for 语句里的表达式。表达式--u 从 u 当中减去 1，得到的结果-1 并不满足无符号数的要求，此时像所有表示范围之外的其他数字一样，-1 被自动地转换成一个合法的无符号数。假设 int 类型占 32 位，则当 u 等于 0 时，--u 的结果将会是 4294967295。

一种解决的办法是，用 while 语句来代替 for 语句，因为前者让我们能够在输出变量之前（而非之后）先减去 1：

```
unsigned u = 11; // 确定要输出的最大数，从比它大 1 的数开始
while (u > 0){
    --u;          // 先减 1，这样最后一次迭代时就会输出 0
    std::cout << u << std::endl;
}
```

改写后的循环先执行对循环控制变量减 1 的操作，这样最后一次迭代时，进入循环的 u 值为 1。此时将其减 1，则这次迭代输出的数就是 0；下一次再检验循环条件时，u 的值等

于 0 而无法再进入循环。因为我们要先做减 1 的操作，所以初始化 u 的值应该比要输出的最大值大 1。这里，u 初始化为 11，输出的最大数是 10。

提示：切勿混用带符号类型和无符号类型

如果表达式里既有带符号类型又有无符号类型，当带符号类型取值为负时会出现异常结果，这是因为带符号数会自动地转换成无符号数。例如，在一个形如 a*b 的式子中，如果 a = -1，b = 1，而且 a 和 b 都是 int，则表达式的值显然为-1。然而，如果 a 是 int，而 b 是 unsigned，则结果须视在当前机器上 int 所占位数而定。在我们的环境里，结果是 4294967295。

2.1.2 节练习

38

练习 2.3：读程序写结果。

```
unsigned u = 10, u2 = 42;
std::cout << u2 - u << std::endl;
std::cout << u - u2 << std::endl;

int i = 10, i2 = 42;
std::cout << i2 -i<< std::endl;
std::cout << i - i2<< std::endl;
std::cout << i - u<< std::endl;
std::cout << u - i<< std::endl;
```

练习 2.4：编写程序检查你的估计是否正确，如果不正确，请仔细研读本节直到弄明白问题所在。

2.1.3 字面值常量

一个形如 42 的值被称作**字面值常量**（literal），这样的值一望而知。每个字面值常量都对应一种数据类型，字面值常量的形式和值决定了它的数据类型。

整型和浮点型字面值

我们可以将整型字面值写作十进制数、八进制数或十六进制数的形式。以 0 开头的整数代表八进制数，以 0x 或 0X 开头的代表十六进制数。例如，我们能用下面的任意一种形式来表示数值 20：

```
20 /* 十进制 */      024 /* 八进制 */      0x14 /* 十六进制 */
```

整型字面值具体的数据类型由它的值和符号决定。默认情况下，十进制字面值是带符号数，八进制和十六进制字面值既可能是带符号的也可能是无符号的。十进制字面值的类型是 int、long 和 long long 中尺寸最小的那个（例如，三者当中最小是 int），当然前提是这种类型要能容纳下当前的值。八进制和十六进制字面值的类型是能容纳其数值的 int、unsigned int、long、unsigned long、long long 和 unsigned long long 中的尺寸最小者。如果一个字面值连与之关联的最大的数据类型都放不下，将产生错误。类型 short 没有对应的字面值。在表 2.2（第 37 页）中，我们将以后缀代表相应的字面值类型。

尽管整型字面值可以存储在带符号数据类型中，但严格来说，十进制字面值不会是负

数。如果我们使用了一个形如-42的负十进制字面值，那个负号并不在字面值之内，它的作用仅仅是对字面值取负值而已。

浮点型字面值表现为一个小数或以科学计数法表示的指数，其中指数部分用E或e标识：

```
3.14159      3.14159E0      0.      0e0      .001
```

39 默认的，浮点型字面值是一个double，我们可以使用表2.2（第37页）中的后缀来表示其他浮点型。

字符和字符串字面值

由单引号括起来的一个字符称为char型字面值，双引号括起来的零个或多个字符则构成字符串型字面值。

```
'a'       // 字符字面值
"Hello World!"   // 字符串字面值
```

字符串字面值的类型实际上是由常量字符构成的数组（array），该类型将在3.5.4节（第109页）介绍。编译器在每个字符串的结尾处添加一个空字符（'\0'），因此，字符串字面值的实际长度要比它的内容多1。例如，字面值'A'表示的就是单独的字符A，而字符串"A"则代表了一个字符的数组，该数组包含两个字符：一个是字母A、另一个是空字符。

如果两个字符串字面值位置紧邻且仅由空格、缩进和换行符分隔，则它们实际上是一个整体。当书写的字符串字面值比较长，写在一行里不太合适时，就可以采取分开书写的方式：

```
// 分多行书写的字符串字面值
std::cout << "a really, really long string literal "
             "that spans two lines" << std::endl;
```

转义序列

有两类字符程序员不能直接使用：一类是**不可打印**（nonprintable）的字符，如退格或其他控制字符，因为它们没有可视的图符；另一类是在C++语言中有特殊含义的字符（单引号、双引号、问号、反斜线）。在这些情况下需要用到**转义序列**（escape sequence），转义序列均以反斜线作为开始，C++语言规定的转义序列包括：

换行符	\n	横向制表符	\t	报警（响铃）符	\a
纵向制表符	\v	退格符	\b	双引号	\"
反斜线	\\	问号	\?	单引号	\'
回车符	\r	进纸符	\f		

在程序中，上述转义序列被当作一个字符使用：

```
std::cout << '\n';            // 转到新一行
std::cout << "\tHi!\n";       // 输出一个制表符，输出"Hi!"，转到新一行
```

我们也可以使用泛化的转义序列，其形式是\x后紧跟1个或多个十六进制数字，或者\后紧跟1个、2个或3个八进制数字，其中数字部分表示的是字符对应的数值。假设使用的是Latin-1字符集，以下是一些示例：

```
\7 （响铃）    \12 （换行符）      \40（空格）
\0 （空字符）  \115 （字符M）      \x4d （字符M）
```

40 我们可以像使用普通字符那样使用C++语言定义的转义序列：

```
std::cout << "Hi \x4dO\115!\n";   //输出Hi MOM!，转到新一行
```

```
std::cout << '\115' << '\n';        //输出M，转到新一行
```

注意，如果反斜线\后面跟着的八进制数字超过 3 个，只有前 3 个数字与\构成转义序列。例如，"\1234"表示 2 个字符，即八进制数 123 对应的字符以及字符 4。相反，\x 要用到后面跟着的所有数字，例如，"\x1234"表示一个 16 位的字符，该字符由这 4 个十六进制数所对应的比特唯一确定。因为大多数机器的 char 型数据占 8 位，所以上面这个例子可能会报错。一般来说，超过 8 位的十六进制字符都是与表 2.2 中某个前缀作为开头的扩展字符集一起使用的。

指定字面值的类型

通过添加如表 2.2 中所列的前缀和后缀，可以改变整型、浮点型和字符型字面值的默认类型。

```
L'a'          // 宽字符型字面值，类型是 wchar_t
u8"hi!"       // utf-8 字符串字面值（utf-8 用 8 位编码一个 Unicode 字符）
42ULL         // 无符号整型字面值，类型是 unsigned long long
1E-3F         // 单精度浮点型字面值，类型是 float
3.14159L      // 扩展精度浮点型字面值，类型是 long double
```

Best Practices

当使用一个长整型字面值时，请使用大写字母 L 来标记，因为小写字母 l 和数字 1 太容易混淆了。

表 2.2：指定字面值的类型

字符和字符串字面值		
前缀	含义	类型
u	Unicode 16 字符	char16_t
U	Unicode 32 字符	char32_t
L	宽字符	wchar_t
u8	UTF-8（仅用于字符串字面常量）	char

整型字面值		浮点型字面值	
后缀	最小匹配类型	后缀	类型
u or U	unsigned	f 或 F	float
l or L	long	l 或 L	long double
ll or LL	long long		

对于一个整型字面值来说，我们能分别指定它是否带符号以及占用多少空间。如果后缀中有 U，则该字面值属于无符号类型，也就是说，以 U 为后缀的十进制数、八进制数或十六进制数都将从 unsigned int、unsigned long 和 unsigned long long 中选择能匹配的空间最小的一个作为其数据类型。如果后缀中有 L，则字面值的类型至少是 long；如果后缀中有 LL，则字面值的类型将是 long long 和 unsigned long long 中的一种。显然我们可以将 U 与 L 或 LL 合在一起使用。例如，以 UL 为后缀的字面值的数据类型将根据具体数值情况或者取 unsigned long，或者取 unsigned long long。

41

布尔字面值和指针字面值

true 和 false 是布尔类型的字面值：

```
bool test = false;
```

nullptr 是指针字面值，2.3.2 节（第 47 页）将有更多关于指针和指针字面值的介绍。

2.1.3 节练习

练习 2.5：指出下述字面值的数据类型并说明每一组内几种字面值的区别：

```
(a) 'a', L'a', "a", L"a"
(b) 10, 10u, 10L, 10uL, 012, 0xC
(c) 3.14, 3.14f, 3.14L
(d) 10, 10u, 10., 10e-2
```

练习 2.6：下面两组定义是否有区别，如果有，请叙述之：

```
int month = 9, day = 7;
int month = 09, day = 07;
```

练习 2.7：下述字面值表示何种含义？它们各自的数据类型是什么？

```
(a) "Who goes with F\145rgus?\012"
(b) 3.14e1L      (c) 1024f      (d) 3.14L
```

练习 2.8：请利用转义序列编写一段程序，要求先输出 2M，然后转到新一行。修改程序使其先输出 2，然后输出制表符，再输出 M，最后转到新一行。

2.2 变量

变量提供一个具名的、可供程序操作的存储空间。C++中的每个变量都有其数据类型，数据类型决定着变量所占内存空间的大小和布局方式、该空间能存储的值的范围，以及变量能参与的运算。对 C++程序员来说，“变量（variable）”和“对象（object）”一般可以互换使用。

2.2.1 变量定义

变量定义的基本形式是：首先是**类型说明符**（type specifier），随后紧跟由一个或多个
42 变量名组成的列表，其中变量名以逗号分隔，最后以分号结束。列表中每个变量名的类型都由类型说明符指定，定义时还可以为一个或多个变量赋初值：

```
int sum = 0, value,  // sum、value 和 units_sold 都是 int
    units_sold = 0;  // sum 和 units_sold 初值为 0
Sales_item item;     // item 的类型是 Sales_item（参见 1.5.1 节，第 17 页）
// string 是一种库类型，表示一个可变长的字符序列
std::string book("0-201-78345-X");//book 通过一个 string 字面值初始化
```

book 的定义用到了库类型 std::string，像 iostream（参见 1.2 节，第 6 页）一样，string 也是在命名空间 std 中定义的，我们将在第 3 章中对 string 类型做更详细的介绍。眼下，只需了解 string 是一种表示可变长字符序列的数据类型就可以了。C++库提供了几种初始化 string 对象的方法，其中一种是把字面值拷贝给 string 对象（参见 2.1.3 节，第 36 页），因此在上例中，book 被初始化为 0-201-78345-X。

术语：何为对象？

C++程序员们在很多场合都会使用**对象**（object）这个名词。通常情况下，对象是指一块能存储数据并具有某种类型的内存空间。

一些人仅在与类有关的场景下才使用“对象”这个词。另一些人则已把命名的对象和未命名的对象区分开来，他们把命名了的对象叫做变量。还有一些人把对象和值区分开来，其中对象指能被程序修改的数据，而值（value）指只读的数据。

本书遵循大多数人的习惯用法，即认为对象是具有某种数据类型的内存空间。我们在使用对象这个词时，并不严格区分是类还是内置类型，也不区分是否命名或是否只读。

初始值

当对象在创建时获得了一个特定的值，我们说这个对象被**初始化**（initialized）了。用于初始化变量的值可以是任意复杂的表达式。当一次定义了两个或多个变量时，对象的名字随着定义也就马上可以使用了。因此在同一条定义语句中，可以用先定义的变量值去初始化后定义的其他变量。

```
//正确：price 先被定义并赋值，随后被用于初始化 discount
double price = 109.99, discount = price * 0.16;
//正确：调用函数 applyDiscount，然后用函数的返回值初始化 salePrice
double salePrice = applyDiscount(price, discount);
```

在C++语言中，初始化是一个异常复杂的问题，我们也将反复讨论这个问题。很多程序员对于用等号=来初始化变量的方式倍感困惑，这种方式容易让人认为初始化是赋值的一种。事实上在C++语言中，初始化和赋值是两个完全不同的操作。然而在很多编程语言中二者的区别几乎可以忽略不计，即使在C++语言中有时这种区别也无关紧要，所以人们特别容易把二者混为一谈。需要强调的是，这个概念至关重要，我们也将在后面不止一次提及这一点。 43

初始化不是赋值，初始化的含义是创建变量时赋予其一个初始值，而赋值的含义是把对象的当前值擦除，而以一个新值来替代。

列表初始化

C++语言定义了初始化的好几种不同形式，这也是初始化问题复杂性的一个体现。例如，要想定义一个名为 units_sold 的 int 变量并初始化为 0，以下的4条语句都可以做到这一点：

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
```

作为C++11新标准的一部分，用花括号来初始化变量得到了全面应用，而在此之前，这种初始化的形式仅在某些受限的场合下才能使用。出于3.3.1节（第88页）将要介绍的原因，这种初始化的形式被称为**列表初始化**（list initialization）。现在，无论是初始化对象还是某些时候为对象赋新值，都可以使用这样一组由花括号括起来的初始值了。 C++ 11

当用于内置类型的变量时，这种初始化形式有一个重要特点：如果我们使用列表初始化且初始值存在丢失信息的风险，则编译器将报错：

```
long double ld = 3.1415926536;
int a{ld}, b = {ld};      // 错误：转换未执行，因为存在丢失信息的危险
int c(ld), d = ld;        // 正确：转换执行，且确实丢失了部分值
```

使用 long double 的值初始化 int 变量时可能丢失数据，所以编译器拒绝了 a 和 b 的初始化请求。其中，至少 ld 的小数部分会丢失掉，而且 int 也可能存不下 ld 的整数部分。

刚刚所介绍的看起来无关紧要，毕竟我们不会故意用 long double 的值去初始化 int 变量。然而，像第 16 章介绍的一样，这种初始化有可能在不经意间发生。我们将在 3.2.1 节（第 76 页）和 3.3.1 节（第 88 页）对列表初始化做更多介绍。

默认初始化

如果定义变量时没有指定初值，则变量被**默认初始化**（default initialized），此时变量被赋予了“默认值”。默认值到底是什么由变量类型决定，同时定义变量的位置也会对此有影响。

44

如果是内置类型的变量未被显式初始化，它的值由定义的位置决定。定义于任何函数体之外的变量被初始化为 0。然而如 6.1.1 节（第 185 页）所示，一种例外情况是，定义在函数体内部的内置类型变量将**不被初始化**（uninitialized）。一个未被初始化的内置类型变量的值是未定义的（参见 2.1.2 节，第 33 页），如果试图拷贝或以其他形式访问此类值将引发错误。

每个类各自决定其初始化对象的方式。而且，是否允许不经初始化就定义对象也由类自己决定。如果类允许这种行为，它将决定对象的初始值到底是什么。

绝大多数类都支持无须显式初始化而定义对象，这样的类提供了一个合适的默认值。例如，以刚刚所见为例，string 类规定如果没有指定初值则生成一个空串：

```
std::string empty;    // empty 非显式地初始化为一个空串
Sales_item item;      // 被默认初始化的 Sales_item 对象
```

一些类要求每个对象都显式初始化，此时如果创建了一个该类的对象而未对其做明确的初始化操作，将引发错误。

定义于函数体内的内置类型的对象如果没有初始化，则其值未定义。类的对象如果没有显式地初始化，则其值由类确定。

2.2.1 节练习

练习 2.9：解释下列定义的含义。对于非法的定义，请说明错在何处并将其改正。

```
(a) std::cin >> int input_value;     (b) int i = { 3.14 };
(c) double salary = wage = 9999.99; (d) int i = 3.14;
```

练习 2.10：下列变量的初值分别是什么？

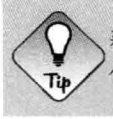
```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

45

提示：未初始化变量引发运行时故障

未初始化的变量含有一个不确定的值，使用未初始化变量的值是一种错误的编程行为并且很难调试。尽管大多数编译器都能对一部分使用未初始化变量的行为提出警告，但严格来说，编译器并未被要求检查此类错误。

使用未初始化的变量将带来无法预计的后果。有时我们足够幸运，一访问此类对象程序就崩溃并报错，此时只要找到崩溃的位置就很容易发现变量没被初始化的问题。另外一些时候，程序会一直执行完并产生错误的结果。更糟糕的情况是，程序结果时对时错、无法把握。而且，往无关的位置添加代码还会导致我们误以为程序对了，其实结果仍旧有错。



建议初始化每一个内置类型的变量。虽然并非必须这么做，但如果我们不能确保初始化后程序安全，那么这么做不失为一种简单可靠的方法。

2.2.2 变量声明和定义的关系

为了允许把程序拆分成多个逻辑部分来编写，C++语言支持**分离式编译**（separate compilation）机制，该机制允许将程序分割为若干个文件，每个文件可被独立编译。

如果将程序分为多个文件，则需要有在文件间共享代码的方法。例如，一个文件的代码可能需要使用另一个文件中定义的变量。一个实际的例子是 `std::cout` 和 `std::cin`，它们定义于标准库，却能被我们写的程序使用。

为了支持分离式编译，C++语言将声明和定义区分开来。**声明**（declaration）使得名字为程序所知，一个文件如果想使用别处定义的名字则必须包含对那个名字的声明。而**定义**（definition）负责创建与名字关联的实体。

变量声明规定了变量的类型和名字，在这一点上定义与之相同。但是除此之外，定义还申请存储空间，也可能会为变量赋一个初始值。

如果想声明一个变量而非定义它，就在变量名前添加关键字 `extern`，而且不要显式地初始化变量：

```
extern int i;     // 声明 i 而非定义 i
int j;            // 声明并定义 j
```

任何包含了显式初始化的声明即成为定义。我们能给由 `extern` 关键字标记的变量赋一个初始值，但是这么做也就抵消了 `extern` 的作用。`extern` 语句如果包含初始值就不再是声明，而变成定义了：

```
extern double pi = 3.1416; // 定义
```

在函数体内部，如果试图初始化一个由 `extern` 关键字标记的变量，将引发错误。

变量能且只能被定义一次，但是可以被多次声明。

声明和定义的区别看起来也许微不足道，但实际上却非常重要。如果要在多个文件中使用同一个变量，就必须将声明和定义分离。此时，变量的定义必须出现在且只能出现在一个文件中，而其他用到该变量的文件必须对其进行声明，却绝对不能重复定义。

关于 C++语言对分离式编译的支持我们将在 2.6.3 节（第 67 页）和 6.1.3 节（第 186

46

页）中做更详细的介绍。

2.2.2 节练习

练习 2.11：指出下面的语句是声明还是定义：

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern int iz;
```

关键概念：静态类型

C++是一种静态类型（statically typed）语言，其含义是在编译阶段检查类型。其中，检查类型的过程称为类型检查（type checking）。

我们已经知道，对象的类型决定了对象所能参与的运算。在C++语言中，编译器负责检查数据类型是否支持要执行的运算，如果试图执行类型不支持的运算，编译器将报错并且不会生成可执行文件。

程序越复杂，静态类型检查越有助于发现问题。然而，前提是编译器必须知道每一个实体对象的类型，这就要求我们在使用某个变量之前必须声明其类型。

2.2.3 标识符

C++的标识符（identifier）由字母、数字和下画线组成，其中必须以字母或下画线开头。标识符的长度没有限制，但是对大小写字母敏感：

```
// 定义 4 个不同的 int 变量
int somename, someName, SomeName, SOMENAME;
```

如表 2.3 和表 2.4 所示，C++语言保留了一些名字供语言本身使用，这些名字不能被用作标识符。

同时，C++也为标准库保留了一些名字。用户自定义的标识符中不能连续出现两个下画线，也不能以下画线紧连大写字母开头。此外，定义在函数体外的标识符不能以下画线开头。

变量命名规范

变量命名有许多约定俗成的规范，下面的这些规范能有效提高程序的可读性：

47

- 标识符要能体现实际含义。
- 变量名一般用小写字母，如 index，不要使用 Index 或 INDEX。
- 用户自定义的类名一般以大写字母开头，如 Sales_item。
- 如果标识符由多个单词组成，则单词间应有明显区分，如 student_loan 或 studentLoan，不要使用 studentloan。

Best Practices 对于命名规范来说，若能坚持，必将有效。