第二部分

C++ 的专有特征

第一部分讨论了C++的C子集，第二部分讨论C++的专有特征，即不同于C的那些特征。因为大多数C++特征都是被设计来支持面向对象编程（OOP）的，所以第二部分还将讨论面向对象程序设计的理论和特点。我们先从C++概述开始讨论。

第11章　C++语言概述

本章概括了C++的关键概念。C++是面向对象的程序设计语言，它的面向对象的特性是相互密切联系的。在许多实例中，这种密切联系使得在描述C++的一个特性时很难不涉及到另外的特性。而且，C++的面向对象的特性在很多方面是相互关联的，因此要讨论一个特性就必须预先了解一个或多个别的特性。为了声明这个问题，本章首先简单介绍C++最重要的方面，包括它的历史、关键特征、传统与标准C++间的区别。以后各章再详细讨论。

11.1　C++的起源

C++是C的扩充版本。C++对C的扩充首先是由Bjarne Stroustrup于1979年在美国新泽西州Murray Hill的贝尔实验室提出的。最初，他把这种新的语言称为“带类的C”，到1983年才改名为C++。

尽管C是世界上最受喜爱和应用最广的专业程序设计语言之一，但C++的发明是必需的。这主要是由程序设计的复杂性所决定的。这些年来，计算机程序变得越来越大、越来越复杂。即使C语言是相当好的程序设计语言，也有其局限性。在C中，一旦程序代码达到25 000至100 000行，它就会变得十分复杂，也就很难全面掌握它了。面C++的目的正是要扫清这个障碍。C++的本质就是让程序员理解和管理更大、更复杂的程序。

Stroustrup对C作了许多的补充以支持面向对象的程序设计（OOP）。下一节对“面向对象的程序设计”有精确解释。Stroustrup宣称C++的某些面向对象的特点受到另一种所谓Simula67的面向对象语言的启发。所以，C++代表着两种强大的程序设计方法的结合。

自问世以来，C++经历了三次主要修改，每次都增加并改变了这种语言。第一次是在1985年，第二次是在1989年，第三次是在开始制定C++标准时。几年前，对C++标准的制定工作开始，一个联合的ANSI((American National Standards Institute，美国国家标准协会)和ISO(International Standards Organization，国际标准化组织)标准委员会成立。所提出的标准的第一稿写于1994年1月25日。在那个草案中，ANSI/ISO C++委员会保留了Stroustrup定义的所有特性，并另外增加了一些新的特性。总体来讲，这个最初的草案反应了C++当时的状态。

在C++标准的第一稿完成后不久，发生了一件事，使得这种语言得以大大扩展：Alexander Stepanov创建了标准模板库（Standard Template Library，STL）。STL是一组通用程序，可以用来操作数据。标准模板库很强大、也很美妙，但是相当大。在第一版草案后，委员会投票表决在C++规范中包含STL。STL的增加使C++的作用范围扩展到其最初的定义之外。最重要的是：标准模板库的包含减慢了C++标准化的工作。

标准化过程相当缓慢，在这个过程中，增加了许多新特征，也进行了许多小的修改。事实上，由C++委员会定义的C++版本远比Stroustrup最初设计的要大得多，也复杂得多。最后的草案在1997年11月14日由委员会通过。C++的ANSI/ISO标准在1998年变成了现实，这个C++规范通常称为标准C++。

第二部分

C++的专有特征

第一部分讨论了C++的C子集，第二部分讨论C++的专有特征，即不同于C的那些特征。因为大多数C++特征都是被设计来支持面向对象编程（OOP）的，所以第二部分还将讨论面向对象程序设计的理论和特点。我们先从C++概述开始讨论。

制接口。

不管为了什么目的，对象是用户定义的类型的变量。虽然把连接代码和数据的对象想做一个变量可能显得有点怪，但是，在面向对象的程序设计中，情况就是这样。定义一个新对象实际上就是创建一种新的数据类型。这个数据类型的每个特定实例是一个复合变量。

11.2.2 多态

面向对象的程序设计语言支持“多态”，即“一个接口，多个方法”。简言之，多态是一种属性，这种属性允许一个接口来控制对一类行为的访问。所选的特定行为由环境的属性决定。多态在现实世界的一个实例是恒温器。不管房子里有什么炉子（煤气、石油、电等），恒温器同样工作。这样，恒温器（接口）与炉子（方法）的种类无关。例如，如果想要华氏70度的温度，设置恒温器到70度，不用考虑提供热量的火炉的类型。

这个规则同样适用于程序设计。例如，可能有一个程序定义了三种不同类型的栈。一个用于整数值，一个用于字符值，一个用于浮点值。因为多态，你可以定义一组名字，即push( )和pop( )用于所有三个堆栈。在程序中，我们将创建三个函数，一个堆栈一个，但是函数名相同。编译器根据所存储的数据自动选择正确的函数。因此，堆栈的接口，即函数push( )和pop( )是相同的，不管所用的堆栈的类型是什么。这三个函数为每种数据类型定义了具体的实现（方法）。

多态性允许同样的接口用于同一类行为，以降低复杂性，根据不同情况选择特定的行为是编译器的事情，程序员不必选择，只要记住和使用通用接口就可以了。第一种面向对象的程序设计语言是解释型的，所以在运行时当然是支持多态性的。然而，C++是编译型语言，所以它既支持编译时的多态性，又支持运行时的多态性。

11.2.3 继承

继承是一个对象获得另一个对象的特性的过程。这是非常重要的，因为继承支持分类的概念。绝大多数的知识都可以通过层次分类进行管理。例如，一个鲜红可口的苹果是苹果的一部分，苹果又是水果的一部分，水果又是食物的一部分。如果不使用分类，每个对象就必须显式地定义其所有特性。使用分类，一个对象就只需要定义它独有的性质。只有继承机制才能使一个对象成为一个更一般情况的具体实例。我们将会看到，继承是面向对象设计程序的一个重要方面。

11.3 C++基础

在第一部分中，描述了C++的C子集，并且使用C程序演示了那些特征。从这里开始，所有的例子都将是C++程序，即，它们会使用C++特有的特征。为了便于讨论，从现在起，我们将把C++特有的特征简单地称为“C++特征”。

如果读者有C语言背景，或者如果读者一直在学习本书第一部分中的C子集程序，就应该知道C++程序在某些重要的方面不同于C程序。大多数区别与利用C++的面向对象的功能有关。但是C++程序在其他方面也与C程序有所不同，包括如何执行I/O和包括什么样的头文件。还有，大多数C++程序共享一组常见的特性，这组特性清楚地标识了C++程序。在迁移到C++的面向对象的结构之前，需要理解C++程序的基本要素。

本节描述与所有C++程序有关的几个问题。据此，指出C和早期的C++的一些重要区别。

11.3.1　C++ 程序范例

让我们从下面所示的范例C++程序开始。

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

int main()
{
  int i;

  cout << "This is output.\n"; // this is a single line comment
  /* you can still use C style comments */

  // input a number using >>
  cout << "Enter a number: ";
  cin >> i;

  // now, output a number using <<
  cout << i << " squared is " << i*i << "\n";

  return 0;
}
```

可以看到，这个程序看起来与第一部分中的C子集程序大不相同。一行一行的注释很有用。开始时，包括了头文件<iostream>。这个头文件支持C++风格的I/O操作（C++中的<iostream>对应于C中的stdio.h）。还要注意，名字iostream没有.h扩展名。理由是<iostream>是由标准C++定义的现代头文件之一，现代C++头文件不使用.h扩展名。

程序中的下一行是：

```
using namespace std;
```

这告诉编译器使用std名字空间。名字空间是最近添加到C++中的。一个名字空间创建一个声明区，在这个区中可以放置各种程序元素。名字空间用于帮助组织大型程序的结构。using语句通知编译器你想使用std名字空间，这是整个标准C++库声明所用的名字空间。通过使用std名字空间，可以简化到标准库的访问。本书第一部分的程序仅使用C子集，不需要名字空间语句，因为C的库函数在默认的全局名字空间中是可得到的。

注意： 因为新格式的头文件和名字空间都是在最近才添加到C++中的，所以可能会遇到不使用它们的老的代码。还有，如果在使用一个老式的编译器，它可能不支持它们。关于使用老式编译器的一些叙述可在本章后面找到。

现在看一下下面这行代码。

```
int main()
```

注意main( )中的参数列表是空的。在C++中，这表示main( )没有参数。这与C语言不同，在C中，没有参数的函数必须在它的参数列表中使用void，如下所示：

```
int main(void)
```

这是第一部分的程序中声明main( )的方式。然而，在C++中，使用void是多余且不必要

的。一般的原则是，在C++中，当函数没有参数时，它的参数列表为空，不要求使用void。

下一行代码包含两个C++特征。

```
cout << "This is output.\n"; // this is a single line comment
```

首先，语句

```
cout << "This is output.\n";
```

使得在屏幕上显示This is output，后跟一个回车换行组合。在C++中，<<还有一个作用。它仍然是左移运算符，但是当它用在像本例这样的情况时，它也是一个输出运算符。字cout是一个被链接到屏幕上的标识符（实际上，像C语言一样，C++支持I/O重定向，但是为了讨论方便之故，我们假定cout指向屏幕）。可以使用cout和<<来输出任何内嵌数据类型以及字符串。

注意在C++程序中，仍然可以使用printf( )或其他的C I/O函数。然而，大多数程序员会觉得使用<<更符合C++的精神。还有，虽然在这个例子中使用printf( )输出字符串等价于使用<<，但是C++ I/O系统可被扩展来执行你定义的对对象的操作（某种使用printf( )不能做的事）。

输出表达式后面的是一个C++的单行注释。正如在第10章中所描述的，C++定义了两种类型的注释。第一，可以使用多行注释，其工作方式与在C语言中相同。也可以通过使用//来定义一个单行注释，编译器忽略这样的注释后面的东西，直到该行结束。通常，C++程序员在需要一个较长的注释时使用多行注释，在需要较短的注释时使用单行注释。

接着，程序提示用户键入数字。这个数字是从键盘中用下面这条语句读入的：

```
cin >> i;
```

在C++中，>>运算符仍然保留它右移的含义。然而，当按所示的那样使用时，它也是C++的输入运算符。这条语句把从键盘中读入的值赋给i。标识符cin指的是标准输入设备，通常是键盘。一般来讲，可以使用cin>>来输入一个任何基本数据类型的变量和字符串。

注意： 刚才描述的代码行没有印错。特别地，在i前面没有&符号。当使用基于C的函数如scanf( )来输入信息时，必须显式地把一个指针传给接受信息的变量。这意味着在变量名前面加上“求地址”运算符&。然而，因为>>运算符在C++中实现的方式，不需要（事实上不必使用）&。其这样做的理由将在第13章解释。

尽管这个例子没有显示，但你可以任意使用基于C的任何输入函数，如scanf( )来代替>>。然而，像使用cout一样，大多数程序员认为cin>>更富有C++的精神。

程序中另一个较为有趣的行如下所示：

```
cout << i << "squared is " << i*i << "\n";
```

假定i的值是10，这条语句会导致显示短语10 squared is 100，后跟一个回车换行。如这行所示，可以同时运行几个<<输出操作。

该程序以下面这条语句结束：

```
return 0;
```

这使得0被返回到调用例程（通常是操作系统）。这条语句在C++中的工作方式与其在C中相同。返回0表示程序正常终止，返回非0值表示不正常的程序终止。如果喜欢，也可以使用值EXIT_SUCCESS和EXIT_FAILURE。

11.3.2　关于 I/O 运算符

如前所述，当把运算符<<和>>用于I/O时，它们能够处理C++的任何内嵌数据类型。例如，下面的程序输入一个浮点数、一个双精度数和一个字符串，然后输出它们。

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

int main()
{
  float f;
  char str[80];
  double d;

  cout << "Enter two floating point numbers: ";
  cin >> f >> d;

  cout << "Enter a string: ";
  cin >> str;

  cout << f << " " << d << " " << str;

  return 0;
}
```

运行这个程序的时候，当提示输入字符串时，键入 This is a test。当程序重新显示输入的信息时，只显示This，其余的部分不显示，因为运算符>>在遇到第一个空格符时，就停止输入字符串。所以，程序不会读入“is a test”。这个程序同时也声明可以把几个输入运算放在一个语句里。

C++ I/O运算符可识别第2章描述的全部反斜杠常量。例如，可以接受以下面格式书写的语句：

```
cout << "A\tB\tC";
```

这条语句输出字母 A, B 和 C，中间用制表符分开。

11.3.3　声明局部变量

C程序和C++程序的一个重要的差别就是声明局部变量的时间。在C89中，程序块中所有的局部变量必须在程序块的开始处声明，不能在出现“动作”语句之后再声明局部变量。例如，下面的代码在C89中是错误的：

```
/* Incorrect in C89. OK in C++. */
int f()
{
  int i;
  i = 10;

  int j; /* won't compile as a C program */
  j = i*2;

  return j;
}
```

在C89程序中，这个函数是错误的，因为“赋值”插在i和j的声明之间。然而，当把它作

为C++程序编译时，这段代码完全可以接受。在C++（和在C99）中，局部变量可以在块内的任何地方声明，并不只限于程序块的开始。

下面是前一个程序的另一种形式，程序在需要str前声明了它：

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

int main()
{
  float f;
  double d;
  cout << "Enter two floating point numbers: ";
  cin >> f >> d;

  cout << "Enter a string: ";
  char str[80]; // str declared here, just before 1st use
  cin >> str;

  cout << f << " " << d << " " << str;

  return 0;
}
```

要把所有变量声明在块的开始处还是在第一次使用的地方完全取决于程序员。因为C++对代码和数据具有封装性，所以要在离使用变量最近的地方声明它们而不是放在块的开始。在前面的例子中，变量是分开声明的。但是，很容易构想出更能体现C++这个特点的更复杂例子。

在离使用最近的位置处声明变量有助于避免产生意外的副作用。在函数很大时，在第一次使用的地方声明变量很有利。但是在很小的函数中（像本书中的许多例子），在函数开始处声明变量是合理的。因此，本书只有在函数很大或很复杂时，才把变量声明放在第一次使用它的地方。

把变量声明放在哪里更明智一些，一直存在一些争论。反对者认为，如果把变量声明散布在整个程序块中，那么读者要很快地找到块中所有变量的声明就变得困难了，而且给程序的维护造成了困难。因此，有的程序员不大利用这个特点。本书并不准备在这两种方法之间确定一个立场。但只要合适，特别是在大型函数中，在离使用最近的地方声明变量使程序员更容易创建便于跟踪调试的程序。

11.3.4 不再有“默认为int”

几年前，对C++做了一点修改，这可能影响老式的C++代码和导出到C++的C代码。C89和最初的C++规范都申明当在一个声明中没有明确指定类型时，就假定为int类型。然而，在标准化的过程中，“默认为int”的原则被从C++中删除了。C99也删出了这一原则。然而，仍然存在很多的C和C++代码在使用这一原则。“默认为int”原则的最常见的用法是和函数返回值一起使用。当函数返回一个整数结果时，不明确指定int是最常见的情形。例如，在C89和老式的C++代码中，下面的函数是有效的。

```
func(int i)
{
  return i*i;
}
```

在标准 C++ 中，这个函数必须具有所指定的 int 返回类型，如下所示。

```
int func(int i)
{
  return i*i;
}
```

实际上，为了与老的代码兼容，几乎所有的 C++ 编译器仍然支持“默认为 int”的原则。然而，在新代码中不应该使用这个特征，因为已不再允许它。

11.3.5　bool 数据类型

C++ 定义了一个内嵌的布尔类型，称为 bool。bool 类型的对象仅可以存储 true 或 false，这两者是 C++ 定义的关键字。正如在第一部分中解释的，会发生自动转换，允许把 bool 值转换为整数，反之亦然。具体来讲，任何非 0 值都转换为 true，而 0 转换为 false。也可以出现相反的情况：true 转换为 1，false 转换为 0。因此，0 为假而非 0 为真的基本概念在 C++ 语言中仍然成立。

注意： 尽管 C89 (C++ 的子集)没有定义布尔类型，C99 给 C 语言中添加了一种类型，称为 _Bool，这种类型可以存储值 1 和 0（即真 / 假）。不像 C++，C99 没有把 true 和 false 定义为关键字。因此，C99 定义的 _Bool 与 C++ 定义的 bool 并不兼容。C99 指定 _Bool 而不是 bool 作为关键字的理由是许多以前存在的 C 程序已经定义了它们自己的 bool 版本。通过定义 Boolean 类型为 _Bool，C99 避免了破坏这种以前存在的代码。然而，在此可以在 C++ 和 C99 之间获得兼容性，因为 C99 添加了头文件<stdbool.h>，这个头文件定义了宏 bool, true 和 false。通过包括这个头文件，可以创建与 C99 和 C++ 兼容的代码。

11.4　老的 C++ 与现代 C++

如上所述，在其发展和标准化的过程中，C++ 经历了一个广泛的发展过程。结果，出现了两个版本的 C++。第一个是以 Bjarne Stroustrup 最初的设计为基础的传统版本。第二个是标准 C++，标准 C++ 是由 Stroustrup 和 ANSI/ISO 标准化委员会创建的。虽然这两个版本的 C++ 其核心内容非常相似，但是标准 C++ 包含几个传统 C++ 中没有的增强。因此，标准 C++ 从本质上讲是传统 C++ 的一个超集。

本书描述了标准 C++。这是由 ANSI/ISO 标准化委员会定义的、由所有现代 C++ 编译器实现的 C++ 版本。本书中的代码反映了标准 C++ 所提倡的现代编码风格和实践。然而，如果你正在使用一个较老的编译器，它将可能不会接受本书中的所有程序。原因如下：在标准化的过程中，ANSI/ISO 委员会给这种语言增加了许多新特征，在定义这些特征时，它们没有被编译器开发商实现。当然，在新特征添加到语言中和它在商用编译器中实现之间有一个滞后期。因为这些特征是在一段时间（若干年）内逐步添加到 C++ 语言中的，老的编译器可能不支持它们中的一个或多个。这是非常重要的，因为近期的添加到 C++ 语言中的两个特征影响了你将编写的每个程序——甚至是最简单的。如果你正在使用老式的编译器，而这个编译器不接受这些新特征，不必担心。下面描述一个很容易的解决办法。

在老式和现代代码之间的关键区别涉及两个特征：新的头文件和名字空间语句。要理解这些区别，让我们先看看一个小的什么也不做的 C++ 程序的两种版本。如下所示的第一个版本反

映了使用老的编码样式编写C++程序的方法。

```
/*
  An old-style C++ program.
*/

#include <iostream.h>

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

请特别注意#include语句。它包含文件iostream.h, 不是头文件<iostream>。还要注意没有名字空间语句。

下面是使用现代样式的程序版本。

```
/*
   A modern-style C++ program that uses
   the new-style headers and a namespace.
*/
#include <iostream>
using namespace std;

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

这个版本使用C++风格的头文件并指定了名字空间。前面已提到过这两个特征。现在让我们仔细讨论一下它们。

11.4.1 新的C++头文件

我们知道，当在程序中使用库函数时，必须包含它的头文件。使用#include语句可以做到这一点。例如，在C语言中，要为I/O函数包含头文件，使用下面这样的语句包含stdio.h：

```
#include <stdio.h>
```

其中，stdio.h是I/O函数使用的文件的名字，前面的语句导致那个文件被包含在程序中。关键是这个#include语句通常包含一个文件。

当C++刚发明时，以及其后几年内，它使用与C语言同样风格的头文件。事实上，为了向后兼容的缘故，标准C++仍然支持你所创建的C风格的头文件。然而，标准C++创建了一种新的头文件，为标准C++库所用。新的头文件没有指定文件名。相反，它们简单地指定标准标识符（可以被编译器映射到文件），尽管不需要如此。新的C++头文件是一种抽象，简单地保证声明了由C++库所要求的合适的原型和定义。

因为新的头文件不是文件名，它们没有.h扩展名。它们由包含于尖括号的文件名组成。例如，下面是标准C++支持的一些新格式的头文件。

```
<iostream>  <fstream>  <vector>  <string>
```

使用 #include 语句来包含新格式的头文件。惟一的区别是，新格式的头文件不必表示文件名。

因为 C++ 包含了整个 C 函数库，它仍然支持与那个库相关联的标准 C 格式的头文件。即，仍可得到像 stdio.h 或 ctype.h 这样的头文件。然而，标准 C++ 也定义了新格式的头文件，可取代这些头文件。C 标准头文件的 C++ 版本只是简单地在文件名前加上 "c" 前缀并去掉 .h。例如，math.h 的 C++ 新格式的头文件是<cmath>，string.h 则是<cstring>。尽管现在在使用 C 库函数时，允许包含 C 格式的头文件，标准 C++ 并不推荐这种方法。因此，自现在开始，本书在所有的 #include 语句中将使用新格式的 C++ 头文件。如果对于 C 函数库，你的编译器不支持新格式的头文件，那么简单的用老格式的类 C 头文件代替。

因为新格式的头文件是近期添加到 C++ 中的，所以许多老的程序都没有使用它。这些程序使用 C 格式的头文件，其中指定了文件名。正如老格式的程序所示的，传统的包含 I/O 头文件的方式如下所示：

```
#include <iostream.h>
```

这使得文件 iostream.h 被包含在程序中。通常，老格式的头文件将使用与它对应的新格式头文件同样的名字，另外增加了 .h。

在写作本书时，所有的 C++ 编译器都支持老格式的头文件，但是，老格式的头文件已过时了，并不推荐在新程序中使用它们。这就是本书不使用它们的原因。

记住：虽然老式的头文件在现有的 C++ 代码中很常见，但它们已过时了。

11.4.2 名字空间

当在程序中包含新格式的头文件时，那个头文件的内容被包含在 std 名字空间中。一个名字空间是一个声明性的区域。名字空间的目的是定位（localize）标识符的名字，以避免名字冲突。在一个名字空间中声明的元素与在另一个名字空间中声明的元素是相分离的。最初，C++ 库函数等的名字被简单地放到全局名字空间中（像 C 中一样）。然而，随着新格式头文件的发明，这些头文件的内容被放到了 std 名字空间中。本书后面我们将仔细讨论名字空间。现在，不需要担心它们，因为下面这条语句：

```
using namespace std;
```

使 std 名字空间成为可见的（即，它把 std 放到了全局名字空间中）。在编译完这条语句后，在使用老格式和新格式的头文件之间没有什么区别了。

最后要注意一点，为了兼容性的缘故，当 C++ 程序包含了 C 头文件，如 stdio.h 时，它的内容被放到了全局名字空间中。这允许 C++ 编译器编译 C 子集程序。

11.4.3 使用老的编译器

如上所述，名字空间和新的头文件都是近期添加到 C++ 语言中的，是在标准化的过程中添加的。当所有的新 C++ 编译器支持这些特征时，老的编译器不会支持。此时，当编译器尝试编译本书范例程序的前两行时，它将报告一个或更多的错误。如果是这样，有一个容易的解决办法：简单地使用一个老格式的头文件并删除 namespace 语句。即，用：

```
#include <iostream.h>
```

取代

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

这一改变把一个现代程序转换成一个老式的程序。因为老式的头文件把它的所有内容读到全局名字空间中，所以不需要 namespace 语句了。

另外，在现在及未来的几年里，你将会看到许多C++程序使用老格式的头文件并且没有包含 using 语句．C++ 编译器能够很好地编译它们。然而，对于新程序，应该使用现代格式，因为它是符合C++标准的惟一程序格式。虽然在许多年内老的程序将继续得到支持，从技术上讲它们是不合适的。

11.5 C++的类

本节介绍C++的一个最重要的特征：类（class）。在C++中，要创建一个对象，首先必须用关键字 class 定义它的一般形式。类和结构（struct）在语法上很相似。下面的类定义了一个用于创建堆栈的类型 stack：

```
#define SIZE 100
// This creates the class stack.
class stack {
  int stck[ SIZE] ;
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};
```

一个类可能包含私有部分，也可能包含公有部分。默认时，类中定义的所有项目都是私有的。例如，变量 stck 和 tos 就是私有的。这意味着，任何不是类的成员的函数不能访问这些变量。这是获得封装性的一种方法，即把这些变量保持为私有，从而牢牢控制对数据项的访问。同样也可以定义只能被该类的其他成员调用的私有函数（尽管在这个例子中并没有显示）。

要把类的某些部分定义为公有的（既可以被程序的其他部分访问），就必须在声明前加上关键字 public。关键字 public 之后声明的所有变量和函数可以被程序的所有其他函数访问。从根本上讲，程序的其他部分总是通过公有函数访问对象。这里要说明的是，尽管可以有公有变量，但原则上应该尽量少用或不用。相反，应该使所有数据成为私有的并且通过公有函数控制对数据的访问。另外，要注意关键字 public 后跟了一个冒号。

因为函数 init( )，push( )和 pop( )是类 stack 的一部分，所以称为成员函数，变量 stck 和 tos 称为成员变量（成数据成员）。记住，对象是数据和代码的联合体。类的私有部分只能被类的成员函数访问，因此，只有 init( )，push( )和 pop( )才能访问 stck 和 tos。

一旦定义了一个类，就可以用类名来创建这种类型的对象。实际上，类名变成了新的数据类型限定符。例如，下面的语句创建了一个 stack 类型的对象 mystack：

```
stack mystack;
```

当声明一个类的对象时，你正在创建那个类的一个实例。在本例中，mystack是stack的一个实例。也可以在定义类的时候通过把名字放在结束大括号之后来创建变量，就像定义结构时所做的那样。

回顾一下，在C++中，一个类创建了一种可以用来创建这种类型对象的新数据类型。所以，一个对象就是类的一个实例，如同某个变量是整型的一个实例一样。换句话说，类是逻辑抽象，而对象才是实体（即对象存在于计算机的内存中）。

类声明的一般形式是：

```
class class-name {
 private data and functions
public:
 public data and functions
} object name list;
```

当然，object name list（对象名列表）可以为空。

在stack的声明中，用到了成员函数的原型。在C++中所有函数都必须给出原型，不可省略。一般来讲，在一个类定义内的成员函数的原型可用做那个函数的原型。

在实际给一个类的成员函数编码时，必须通过用它所属的类名限制函数名，来告诉编译器这个函数是属于哪个类的。例如，下面是一种给函数push( )编码的方法：

```
void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}
```

::称为作用域分辨符。实际上，上述代码告诉编译器push( )的这个形式属于类stack，换句话说，这个push( )在stack的作用域之内。在C++中，几个不同的类可以使用同一个函数名。编译器通过作用域分辨符来判断哪个函数属于哪一个类。

如果要指定不属于该类一部分的成员函数，就必须使用与该类对象相关联的格式：对象名加点运算符再加成员名。这个规则适用于访问数据成员和成员函数。例如，下面的语句为对象stack1调用init( )：

```
stack stack1, stack2;

stack1.init();
```

这里，产生了两个对象（stack1和stack2），并初始化stack1。stack1和stack2是两个分离的对象。这意味着，初始化stack1时并不初始化stack2。stack1和stack2之间的仅有的关系就是它们是同一类型的对象。

在一个类内，一个成员函数可以直接调用另一个成员函数，或者直接调用数据成员而无需使用点运算符。只有不属于该类的代码调用成员函数时，才使用对象名和点运算符。

下面的程序把前面的程序块串在一起并添加了细节，用以演示stack类：

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

#define SIZE 100

// This creates the class stack.
class stack {
  int stck[ SIZE] ;
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};

void stack::init()
{
  tos = 0;
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[ tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
  tos--;
  return stck[ tos];
}

int main()
{
  stack stack1, stack2; // create two stack objects

  stack1.init();
  stack2.init();

  stack1.push(1);
  stack2.push(2);

  stack1.push(3);
  stack2.push(4);

  cout << stack1.pop() << " ";
  cout << stack1.pop() << " ";
```

```
  cout << stack2.pop() << " ";
  cout << stack2.pop() << "\n";

  return 0;
}
```

这个程序的输出如下所示：

```
3 1 4 2
```

记住，对象的私有成员只能被它的成员函数访问。例如，如下语句：

```
stack1.tos = 0; // Error, tos is private.
```

不能放在前面这个程序的 main( )函数中，因为 tos 是私有的。

11.6　函数重载

函数重载是C++获得多态性的途径之一。在C++中，两个或两个以上的函数可以共享同一个名字，只要它们的参数声明是不同的。在这种情况下，共享同一名字的函数称为被重载了，而这个过程称为函数的重载。

为了弄清楚函数重载的重要性，先考虑一下由C子集定义的三个函数：abs( )，labs( )和fabs( )。函数 abs( )返回一个整数的绝对值，labs( )返回一个长整数的绝对值，fabs( )返回一个双精度数的绝对值。尽管这三个函数处理几乎完全相同的事情，但是在C语言中却要用三个稍有不同的名字来表示三个基本相似的任务。这就使情况从概念上讲变得比实际更为复杂。尽管每个函数的基本概念是相同的，但程序员不得不记住三件事情，不是一件。但是，在C++中，这三个函数可以只用一个名字，如下所示：

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

// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
  cout << abs(-10) << "\n";

  cout << abs(-11.0) << "\n";

  cout << abs(-9L) << "\n";

  return 0;
}

int abs(int i)
{
  cout << "Using integer abs()\n";

  return i<0 ? -i : i;
}
```

```
double abs(double d)
{
  cout << "Using double abs()\n";

  return d<0.0 ? -d : d;
}

long abs(long l)
{
  cout << "Using long abs()\n";

  return l<0 ? -l : l;
}
```

这个程序的输出如下所示：

```
Using integer abs()
10
Using double abs()
11
Using long abs()
9
```

这个程序创建了三个相似但不同的名为abs( )的函数，每一个返回其变元的绝对值。编译器根据变元的类型决定在每一种情况下需要调用哪个函数。重载函数的价值在于允许用一个共同的名字访问相关的函数的集合。因此abs( )代表了将要执行的一般动作，留给编译器的任务就是在一个特定的环境下选择正确的特定方法，程序员只需要记住要执行的一般动作。正是由于多态性，要记住三件事减为只需记住一件事了。这个例子非常平常，但是如果拓展这个概念，就会发现多态性是如何帮助用户管理复杂程序的。

一般来说，要重载一个函数，只需声明它的不同形式，编译器会做好剩下的事情。重载函数时我们必须注意一个重要的限制：重载函数的类型和参数个数必须不同，两个函数仅在它们返回的类型上不同是不够的，它们必须在参数类型和数量上不同（返回类型不能提供足够的信息让编译器判断使用哪个函数）。当然，重载函数的返回类型也可以不同。

下面是另一个使用函数重载的例子：

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

void stradd(char *s1, char *s2);
void stradd(char *s1, int i);

int main()
{
  char str[80];

  strcpy(str, "Hello ");
  stradd(str, "there");
  cout << str << "\n";

  stradd(str, 100);
  cout << str << "\n";
```

```
  return 0;
}

// concatenate two strings
void stradd(char *s1, char *s2)
{
  strcat(s1, s2);
}

// concatenate a string with a "stringized" integer
void stradd(char *s1, int i)
{
  char temp[80];

  sprintf(temp, "%d", i);
  strcat(s1, temp);
}
```

在这个程序中，函数 stradd( )被重载。其中一种形式把两个字符串连接起来（像 strcat( )所做的那样），而另一种形式先把一个整数转换为字符串，再把它添加到一个字符串的尾部。这里，重载被用于创建了一个既能把一个字符串又能把一个整数附加到另一个字符串之后的接口。

可以用相同的名字重载不相关的函数，但最好不要这样做。例如，可以用名字 sqr( )创建返回整数的平方和双精度数的平方根的函数。但是，这两种运算是根本不同的，像这样应用函数重载就使其意图失败了（这是很糟糕的程序设计风格）。实际上，只应该重载那些密切相关的操作。

11.7　运算符重载

多态性在 C++ 中还可以通过运算符重载来获得。在 C++ 中可以用运算符<<和>>来执行控制台 I/O 操作。这些运算符能够完成这些额外的操作，因为它们在头文件<iostream>里被重载了。当一个运算符被重载时，它便有了和某个类相关的附加的含义，但它仍然保持了所有老的含义。

一般来说，可以通过定义对一个具体类来说运算符意味着什么来重载C++的绝大多数运算符。例如，回想一下本章前面介绍过的 stack 类，可以针对类型 stack 的对象重载运算符 +，以便它能够把一个堆栈的内容添加到另一个堆栈中去。但是，+ 仍然保持着它原来针对其他类型数据的含义。

因为实际上运算符重载比函数的重载要复杂一些，所以我们将在第14章中给出运算符重载的例子。

11.8　继承

本章前面提到过，继承是面向对象程序设计语言的主要特征之一。在C++中，继承是通过允许一个类把另一个类放到它的声明中实现的。继承允许建立类的从最一般到最特殊的层次。这个过程要求首先定义一个基类，基类定义了那些由它派生的所有对象共有的性质。基类代表了最一般的描述。由基类派生的类通常称为派生类。一个派生类包括通用基类的所有特点，且增加了派生类特有的性质。为了声明它是如何工作的，下面的例子创建了一个给不同类型的建

筑物分类的类：

开始时，声明building类，如下所示，它用做两个派生类的基类：

```
class building {
  int rooms;
  int floors;
  int area;
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};
```

就这个例子而言，因为所有建筑物都有三个共有的特征：一个或多个房间、一层或几层楼以及总的面积，building类的声明包含了这几个特征。以set开头的成员函数设置了私有数据的值，以get开头的函数返回它们的值。

现在可以用建筑物的这个广义定义创建描述具体类型的建筑物的派生类了。例如，下列是名为house的派生类：

```
// house is derived from building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};
```

注意building是如何被继承的。继承的一般形式是：

```
class derived-class : access base-class {
   // body of new class
}
```

其中，access是可选的。然面，如果存在，它必须是public，private或protected（第12章将进一步讨论这些选项）。这里，所有被继承的类都是public。使用public意味着基类的所有公有成员在派生类中也是公有的。所以，类building的公有成员变成了派生类house的公有成员，类house的成员访问类building的成员函数就像它们是在house中声明的一样。但是，house的成员函数不能访问building的私有元素。这点很重要。尽管house继承了building，但它也只能访问building的公有成员。在这种方法中，继承并不妨碍OOP需要的封装性。

记住：派生类直接访问它自己的成员函数和基类的公有成员。

下列程序演示了继承性。它使用继承创建了building的两个派生类：house和school。

```
#include <iostream>
```

```
using namespace std;

class building {
  int rooms;
  int floors;
  int area;
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};

// house is derived from building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};

// school is also derived from building
class school : public building {
  int classrooms;
  int offices;
public:
  void set_classrooms(int num);
  int get_classrooms();
  void set_offices(int num);
  int get_offices();
};

void building::set_rooms(int num)
{
  rooms = num;
}

void building::set_floors(int num)
{
  floors = num;
}

void building::set_area(int num)
{
  area = num;
}

int building::get_rooms()
{
  return rooms;
```

```
}
int building::get_floors()
{
  return floors;
}
int building::get_area()
{
  return area;
}
void house::set_bedrooms(int num)
{
  bedrooms = num;
}
void house::set_baths(int num)
{
  baths = num;
}
int house::get_bedrooms()
{
  return bedrooms;
}
int house::get_baths()
{
  return baths;
}
void school::set_classrooms(int num)
{
  classrooms = num;
}
void school::set_offices(int num)
{
  offices = num;
}
int school::get_classrooms()
{
  return classrooms;
}
int school::get_offices()
{
  return offices;
}
int main()
{
  house h;
  school s;

  h.set_rooms(12);
```

```
  h.set_floors(3);
  h.set_area(4500);
  h.set_bedrooms(5);
  h.set_baths(3);

  cout << "house has " << h.get_bedrooms();
  cout << " bedrooms\n";

  s.set_rooms(200);
  s.set_classrooms(180);
  s.set_offices(5);
  s.set_area(25000);

  cout << "school has " << s.get_classrooms();
  cout << " classrooms\n";
  cout << "Its area is " << s.get_area();

  return 0;
}
```

这个程序的输出如下所示：

```
house has 5 bedrooms
school has 180 classrooms
Its area is 25000
```

这个程序表明，继承的最主要的优点是能够创建一个可以合并到多个特殊类中去的通用类。这样，每一个对象就可以准确地表示它自己的子类。

在写关于C++的东西时，一般使用基（base）和派生（derived）描述继承关系，但也有用父（parent）和子（child）的。也可以看到术语超类（superclass）和子类（subclass）。

除了提供层次分类的优点以外，继承还通过虚函数机制提供对运行时多态性的支持（详见第16章）。

11.9 构造函数和析构函数

一种常见的情况是，对象的某些部分在使用之前需要初始化。例如，想想本章前面讨论过的stack类。在使用堆栈之前，tos必须设置为0。这是用函数init( )完成的。因为对初始化的要求是如此频繁，所以C++允许对象在创建时初始化它本身。这种自动初始化是通过使用构造函数实现的。

构造函数是一种特殊的、和类同名且为类的成员的函数。例如，下面是一个stack类用构造函数初始化的例子：

```
// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack(); // constructor
  void push(int i);
  int pop();
};
```

注意，构造函数stack( )没有被指定返回类型。在C++中，构造函数不能有任何返回值，且无返回类型。

stack( )构造函数的编码如下：

```
// stack's constructor
stack::stack()
{
  tos = 0;
  cout << "Stack Initialized\n";
}
```

注意，stack初始化的信息只是作为一种说明构造函数的方法输出的。在实际应用中，大多数构造函数并不进行任何输出或输入，而只是完成各种各样的初始化。

一个对象的构造函数在对象创建时被自动调用，这意味着在执行对象声明时才调用它。如果习惯于把声明语句想做被动的，那么C++并非如此。在C++中，声明语句是要执行的语句。这种区别不只是学术上的。要执行构建对象的代码可能是有相当意义的。对于全局或静态局部对象，一个对象的构造函数要被调用一次。对于局部对象，每当遇到对象声明时都要调用构造函数。

与构造函数互补的是析构函数。在许多情况下，当对象被撤销时需要执行一些动作，局部对象是在进入它所在的程序块时创建的，在退出程序块时撤销。全局对象在程序结束时撤销。撤销对象时，自动调用它的析构函数。为什么需要析构函数，有许多理由。例如，一个对象可能需要释放它以前分配的内存，或者关闭已打开的文件。在C++中，处理撤销活动事件的是析构函数。析构函数和构造函数具有相同的名字，只是在前面增加了符号~。例如，下面是stack类以及它的构造函数和析构函数（记住，stack类并不需要析构函数，这里只是为了演示起见）。

```
// This creates the class stack.
class stack {
  int stck[ SIZE] ;
  int tos;
public:
  stack(); // constructor
  ~stack(); // destructor
  void push(int i);
  int pop();
};

// stack's constructor
stack::stack()
{
  tos = 0;
  cout << "Stack Initialized\n";
}

// stack's destructor
stack::~stack()
{
  cout << "Stack Destroyed\n";
}
```

注意，像构造函数一样，析构函数没有返回值。

为了弄清结构函数和析构函数是如何工作的，这里是本章前面讨论过的 stack 程序的新形式。注意，在此不再需要函数 init( )了。

```
// Using a constructor and destructor.
#include <iostream>
using namespace std;

#define SIZE 100

// This creates the class stack.
class stack {
  int stck[ SIZE] ;
  int tos;
public:
  stack(); // constructor
  ~stack(); // destructor
  void push(int i);
  int pop();
};

// stack's constructor
stack::stack()
{
  tos = 0;
  cout << "Stack Initialized\r";
}

// stack's destructor
stack::~stack()
{
  cout << "Stack Destroyed\n";
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[ tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
  tos--;
  return stck[ tos] ;
}
```

```
int main()
{
  stack a, b; // create two stack objects
  a.push(1);
  b.push(2);

  a.push(3);
  b.push(4);

  cout << a.pop() << " ";
  cout << a.pop() << " ";
  cout << b.pop() << " ";
  cout << b.pop() << "\n";

  return 0;
}
```

这个程序显示如下结果：

```
Stack Initialized
Stack Initialized
3 1 4 2
Stack Destroyed
Stack Destroyed
```

11.10 C++的关键字

标准C++中定义了63个关键字，示于表11.1中。与C++的正式语法一起，它们形成了C++编程语言。还有，早期的C++版本定义了overload关键字，但是现在已不用了。记住，C++是区分大小写的语言，它要求所有的关键字都是小写的。

表11.1 C++的关键字

asm	auto	bool	break
case	catch	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

11.11　C++ 程序的一般形式

虽然个人的风格不同，但大多数 C++ 程序员使用下面的一般形式：

```
#includes
base-class declarations
derived class declarations
nonmember function prototypes
int main( )
{
    //...
}
nonmember function definitions
```

然而，对于大多数大型程序，所有的类声明都放在一个头文件中，每个模块都包含它，但程序的一般组织不变。

这一部分的其他章节将详细介绍本章讨论过的 C++ 特性以及 C++ 的其他特性。