第一章　C++概要

C++是C语言的改进版本。C++中包含了C语言的每一部分，不过，C++增加了对面向对象编程(简写为OOP)的支持。另外，与面向对象编程毫无关系，C++还包含许多使C++成为一种"更优秀的"C的改进与特点。伴随很小、很少的不同，C++是C的超集，尽管你对C语言所了解的一切知识都适用于C++，但是理解C++的增强特点对你来说仍需花费相当多的时间与相当大的努力。不过，运用C++进行编程所得到的回报将远远大于你付出的努力。

本章的目的是向你介绍几个C++的重要特点。正如你所知道的，计算机语言的各个要素之间并不是孤立的。相反，它们协同工作，形成完整的语言，这种相互联系在C++中体现得尤其明显。事实上，孤立地讨论C++的某一方面是件困难的事，因为C++的特点是高度一体化，为了有助于克服这个问题，这个概况能够使你理解书中以后讨论的范例。记住：这里讲到的大多数主题在本章的后面都会更系统化地讲解。

除了介绍几个重要的C++特点，本章也要讨论C与C++在编程风格上的不同。C++有几个特点容许你在编写程序的时候拥有更大的灵活性。虽然这些特点中有些与面向对象的编程几乎无关，但是，他们经常可以在大多数C++程序中找到，因此，在本书的前面部分讨论比较适当。

因为发明C++是用来支持面向对象编程的，本章就从描述OOP开始。很快就会看到，C++的许多特点都以这样或那样的方式与OOP相联系。实际上，OOP理论渗透了C++的几个方面。不过，理解C++既可用来编写面向对象的程序，又可用来编写不是面向对象的程序是很重要的。你如何使用C++则完全取决于你自己。

注意：这本书假定你知道如何用C++编译器来编写与执行程序，如果你不知道的话，你就必须参考你的编译器用户手册(因为编译器与编译器不同，所以，不可能在本书中给出所有编译器的编译指令)。因为编程的最好学习方法是实践，所以，你一定要命令自己输入、编译、运行书中的范例——按照它们被提供的顺序。

在你开始之前，关于C++的本质与结构的几点一般注释应当先谈谈。首先，对大多数的情况而言，C++程序看起来很像C程序。就像C程序一样，C++程序从执行main()开始。为了包含命令行参量，C++采用与C同样的argc、argv惯例。C++有与C同样的标准库函数。虽然C++定义了几个专有的头文件，但它也包含C编译器所建立的全部头文件。C++使用与C同样的控制结构。C++与C有相同的内部数据结构。

记住，这本书假定你已经知道了编程语言。简而言之，在你可以学习C++编程之前，你必须能够用语言编程。如果你不了解语言的话，良好的开端便是我的书《Teach Yourself C》，2nd Edition(Berkeley：Osborne/McGraw-Hill，1994)。这本书提供了本书同样的系统方法，并且从整体上覆盖了全部的C语言。

1.1 什么是面向对象的编程

面向对象的编程是一种解决编程任务的新方法。自程序设计的早期以来，就存在许多关于编程的方法论。在编程发展过程中的每一个关键之处，都创造了新的方法来帮助程序员处理越来越复杂的程序。第一个程序是通过处理计算机前面板上的开关完成的。显然，这种方法只适于最小的程序。接着，就发明了汇编语言，它允许编写较长的程序。二十世纪五十年代，又产生了新的编程方式：第一种高级语言(FORTRAN)诞生。

通过采用高级语言，程序员可以编写几千行长的程序。但是，早期采用的编程方法都是特定的，是针对某一事情进行处理的方法。对于较短的程序，这种方法比较适用，但它用于较长的程序时，会发生不可读(并且不可管理)的“实心面代码”。实心面代码的消除在二十世纪六十年代通过结构化编程语言的产生而得以实现。这些结构化编程语言包括 Algol 与 Pascal。不很严格地说，C 语言是一种结构化语言，并且极其类似于你正在使用的，被称为结构化编程的编程类型。结构化编程依赖于适当定义的的控制结构、代码块、无 GoTo 语句(或者至少是最少使用)，和支持递归与局部变量的单独子程序。结构化编程的本质是将程序精减为基本组成。采取结构化编程，平均地讲，程序员可以生成与维护的程序长度可达 50,000 行。

虽然结构化编程具有一些优点，适用于复杂程度中等的程序，但是，当程序到达一定长度之后，它却会在一些方面发生错误。为了能够编写更复杂的程序，一种新的处理编程工作的方法就十分必需了。在这方面探索的结果就是，面向对象编程的产生。OOP 采纳了体现在结构化编程中的最优秀的思想，并将这些思想与最有力的新概念结合起来，使你能够更有效地组织你的程序。面向对象的编程鼓励你将一个问题分解为相互联系的子群体。每一个子群体成为一个包括与其自身相联系的指令与数据的自包含对象。采取这种方法，复杂性减小了，程序员可以管理更大的程序。

所有的 OOP 语言，包括 C++，都具有三个共同的本质特征：封装、多态与继承。下面让我们讨论这些概念。

封装

封装就是将代码与它所操作的数据打包在一起的机理，从而避免外界的交叉引用与误用。在一个面向对象的语言中，代码与数据可以按这样的方式打包：它们形成一个自包含的“黑盒子”。在盒子中就包含了所有必要的数据与代码。当代码与数据以这种形式联系在一起的时候，我们就生成了一个对象。换句话说，对象就是支持封装的一种机制。

在一个对象中，代码、数据或二者一起，既可以是该对象专有的(private)，也可以是公共的(public)。专有数据或代码只对同一对象的其他部分公共并为其访问，也就是说在对象之外的程序不能访问一个对象的专有数据和代码；如果数据和代码是公共的，那么即使是在对象内部定义的，也可以被程序的其他部分访问。典型地，对象的公共部分往往用于为对象的专有部分提供一个受控的接口。

对所有的意图和目的，一个对象就是一个具有用户定义类型的变量。一个融合了代码和数据的对象是变量，这一点看起来似乎有点奇怪，但是，在面向对象的程序设计中，事实就是这样。这次你定义一个新的对象，你就建立了一人新的数据类型，而该数据类型的每一个实例就是一个复合变量。

多态

多态(来自希腊语,意味着"许多形态")是一种性质,它允许一个名字用于两个或两个以上有联系,但技术上截然不同的目的。多态应用于OOP的目的是允许用一个名字来指定动作的一般类。在动作的一般类中,使用到的特定动作将由数据类型确定。例如,在C语言中,并不支持多态,绝对值动作要求三个不同的函数:abs(),labs(),fabs()。这些函数分别计算并返回整数、长整数、浮点值的绝对值。但是,在C++中,支持多态,每个函数都可以被称为abs()(在本章的后面会讲述实现这种abs()调用的一种方法)。被用来调用函数的数据类型决定了实际采用的是哪一特定函数版本。不久你会看到,在C++中,一个函数名用于多个不同目的是可行的,这一功能被称为函数重载。

更一般地讲,多态的概念是一种"一个接口,多种方法"的构想。这就意味着可以针对一组相关的动作设计一个一般接口。但是,特定动作的执行仍然依赖于数据。多态的优势在于:它通过用相同的接口指定动作的一般类来减小复杂性。当特定动作应用于每一个场合时,选择特定动作就是编译器的任务了。你,程序员不需要手工来完成这一工作。你只要记住并使用一般接口就可以了。作为说明前述段落的例子,用绝对值函数的三个名称代替一个会使绝对值的一般操作比实际更为复杂。

多态既可用于函数,也可用于运算。实质上,所有的编程语言都包含有限的多态应用,因为多态是与数学运算联系在一起的,例如,在C++中,符号"+"被用于加整数、长整数、字符串与浮点值。对于这些情况,编译器会自动识别应用了哪一种数学运算。在C++中,你可以扩展这一念到你定义的其它数据类型。这类多态被称为算子重载。

关于多态应当记住的一个关键点是:由于允许建立相关动作的标准接口,你可以处理更复杂的问题。

继承

继承是一种处理方法,通过这一方法,一个对象可以获得另一个对象的特征。更专业化点讲,一个对象可以继承一组一般特征,并将专门指定给它自己的那些特点加到这组一般特征上。继承是很重要的,因为它支持"层次"分类的概念,大多数信息通过层次分类成为可管理的。例如,考虑一下对房子的描述。房子是我们称之为building的一般类。依次类推,building是更一般的类structure的一部分,structure又是我们称之为man-made的更一般的对象类的一部分。在任何情况下,子类总是继承所有那些与父类相关的特征,并将这些特征融合于他自己的定义特征中。如果不采用有序的分类,对于每一个对象,就必须定义所有显然与它自身相关的特征。但是,通过使用继承,描述对象就可以只给出对象所属的一般类(或者类组)和那些使它与其他对象不同的特征。很快你会发现,继承在OOP中扮演了一个非常重要的角色。

举例

1. 封装对于OOP完全不是一个新概念。某种程度上来说,在使用语言的时候,就会获得封装。例如,当你使用了一个库函数,从效果上来讲,你是在使用一个黑盒子子程序,你不可能变更或者影响盒子内部的东西(除非你有意这么做才有可能)。

考虑一下fopen()函数。当它用来打开一个文件时,几个内部变量要生成并加以初始化。就你的程序所关心的东西而言,这些变量是隐含的,并且不可以访问。不过,C++为封装提

供了有把握得多的方法。

2. 在现实世界中，多态的例子相当地普遍，例如，考虑一下你的汽车上的方向盘。无论你的汽车是使用动力方向盘，还是标准的人力方向盘，它们完成的任务都是相同的。这个例子的含义是：无论实际的方向盘操纵机理(方法)如何产生，接口(方向盘)都是同样的。

3. 正如所指出的那样，特性的继承与分类的更一般概念是组织知识的方法的基础。例如，芹菜是 vegetable 类的成员，vegetable 是 Plant 类的一部分。依此类推，植物又是生物体，等等。如果没有层次分类法，知识系统将不可能产生。

练习

1. 考虑一下分类与多态在我们日常生活中扮演重要角色的例子。

1.2 C++控制台 I/O

因为 C++是 C 的一个超集，所以，C 语言的所有组成都包含于 C++语言中，这暗指所有的 C 语言程序通过缺省也都是 C++程序(实际上，这个规则也有一些非常小的例外，这些例外在书的后面会谈到)。因此，可能写一个看起来很象 C 程序的 C++程序。即使本质上没有什么错误，这也意味着你并没有完全利用 C++。

大多数 C++程序员使用对 C++来说是独一无二的风格与特点进行程序设计。这样做的一个原因是它有助于你根据 C++的特点而不是 C 的特点来思考，你可以让任何正在读你的程序的人立刻明白：这个程序是一个 C++程序，而不是 C 程序。

可能 C++程序员使用最普遍的 C++专有特点是它的控制台 I/O 方法。也许你还在使用类似于 printf()与 scanf()之类的函数，但是 C++提供了一种新的、更好的方法，用来执行这类 I/O 操作。在 C++中，输入输出通过 I/O 算子，而不是 I/O 函数来执行。输出操作是<<、输入操作是>>。正如你知道的那样，在 C 语言中，“<<”、“>>”分别对应左移与右移操作。在 C++中，它们仍然保留它们的原始功能(左移与右移)，但是它们也执行输入与输出这个添加的任务。看看这个 C++表达式：

```
cout << "This string is output to the screen. \n";
```

这个语句执行在计算机屏幕上显示指定字符串的功能。Cout 是预定义流，它在 C++程序开始执行的时候，自动与控制台联接。Cout 类似于 C 语言的 stdout。与 C 语言中一样，C++的控制台输入/输出可以被重新定向，但是，就目前我们的讨论而言，仍然假定正在使用控制台。

采用<<输出算子，输出任何 C++基本类型都是可能的。例如，下面的语法输出值 100.99：

```
cout << 100.99;
```

一般来说，对于控制台的输出，使用下面的一般<<运算形式：

```
Cout << expression;
```

其中，expression 是任何有效的 C++表达式——包括其他输出表达式。

要想从键盘输入一个值，可以使用>>输入算子。例如，下面的片断将一个整数值输入 num：

```
int num;
cin >> num;
```

注意，num 前面没有 &。正如你知道的那样，当你用 C 语言 scanf()函数输入某个值的时候，变量一定要将它们的地址传递给函数，以便函数可以接收到用户输入的值。而使用 C++输入运算就完全不是这种情况(这个原因在你学习了更多的 C++知识后，会变得十分清楚)。

一般来讲，为了从键盘输入某个值，可以采用下面的形式：

```
cin >> variable;
```

注意："<<"与">>"的扩展功能是算子重载的例子。

为了使用 C++I/O 算子，你必须将你的头文件包含于你的程序的 iostream.h 中。这是 C++的标准头文件之一，并且由你的 C++编译器提供。

举例

1. 这个程序输出一个字符串，两个整数值与一个双浮点值：

```
#include <iostream.h>
main()
{
  int i,j;
  double d;

  i = 10;
  j = 20;
  d = 99.101;
  cout << "Here are some values:";
  cout << i;
  cout << ' ';
  cout << j;
  cout << ' ';
  cout << d;
  return 0;
}
```

注意：正如这个例子所表示的那样，值 0 从 main 中返回。如果没有技术性的必要，对于程序来说，最好是在程序的结尾将已知的值返回调用过程(通常是操作系统)(本书中所有的程序都遵循这个规则)。如果没有 return 语句，就返回一个未定义的值。

2. 利用一个 I/O 表达式输出多个值是可能的，例如，例 1 中描述的程序的下面这个版本显示了对 I/O 进行编程的一种更有效的方法：

```
#include <iostream.h>
main()
{
  int i,j;
  double d;
  i = 10;
  j = 20;
  d = 99.101;
```

```
  cout << "Here are some values:";
  cout << i << ' ' << j << ' ' << d;
  return 0;
}
```

这里，下面的一行程序

```
cout << i << ' ' << j << ' ' << d;
```

用一个表达式输出几个项目。一般来说，你可以只用一个表达式输出你想输出的任意多的项目。如果这会使你混淆的话，那么只记住：<<输出算子与任何其他C++算子一样，并且是任意长表达式的一部分。

注意，只要有必要你必须清楚地留下项目之间的空格。如果不留下空格的话，在屏幕显示的时候，数据会连在一起。

3. 这个程序为用户提示整数值：

```
#include <iostream.h>
main()
{
  int i;
  cout << "Enter a value:";
  cout >> i;
  cout << "Here's your number:" << i << "\n";
  return 0;
}
```

4. 接下来的这段程序为用户提示整数值浮点值、与字符串。然后，你用一个输入语句读入全部的三个量。

```
#include <iostream.h>
main()
{
  int i;
  float f;
  char s [80];
  cout << "Enter an integer, float, and string:"
  cin >> i >> f >> s;
  cout << "Here' your data:";
  cout << i << ' ' << f << ' ' << s;
  return 0;
}
```

就像举例中所表示的那样，你可以在一个输入语句中，输入你希望输入的多个项目。就象在C语言中一样，单独的数据项目一定要用“空”隔开(空格键、制表键与换行符)。在读入字符串的时候，一旦读入第一个空白字符串，输入就停止。例如，如果你将下面的输出作为前面的程序的输入

10 100.12 This is a test

程序会显示下面内容：

10 100.12 This

这个字符串是不完全的，因为字符串的读入在 this 之后的空白键处停住了。字符串的其余部分被留在了输入缓冲区，等待下一个输入操作(这类似于采用 Scanf()函数、以%S 格式输入字符串)。

5. 当你使用>>时，通过缺省，所有的输入都被作线性缓冲处理。这意味着：直到你按下Enter键，才有信息被传输给你的 C++程序。大多数 C 编译器在用 Scanf()处理问题的时候，也采用线性缓冲输入，用此，线性缓冲对你来说并不应当是新概念。为了看看线性缓冲输入的效果，试试下面的这个程序：

```
#include <iostream.h>
main()
{
  char ch;
  cout << "Enter keys, x to stop.\n";
  do {
    cout << ": ";
    cin >> ch;
  } While (ch ! = 'x');
  return 0;
}
```

在你尝试这个程序的时候，输入了送给程序的每一个字符串相应键之后，你还要按Enter键。

练习

1. 编写一个程序，输入雇员的工作时间与雇员的工资额。然后显示雇员的总收入(一定要给出输入提示)。

2. 编写一个程序，将英尺转换为英寸。提示用户输入英尺，并显示英尺相对应的英寸量。重复这个过程，直到用户输入的英尺值为 0。

3. 下面是一个 C 程序，改写它，使它采用 C++风格的 I/O 语句：

```
/* Convert this C program into C++ style.
   This program computes the lowest common
   denominator.
*/
#include <stdio.h>
main()
{
  int a, b, d, min;

  printf("Enter two numbers: ");
  scanf("%d%d", &a, &b);
  min = a > b ? b : a;
  for(d = 2; d<min; d++)
    if(((a%d)==0) && ((b%d)==0)) break;
  if(d==min) {
    printf("No common denominators\n");
    return 0;
  }
  printf("The lowest common denominator is %d\n", d);
  return 0;
}
```

1.3 C++注释

在C++中,你可能用两种不同的形式包含你的程序注释。第一种方法,你可能采用标准的、类似于C的注释机理。也就是,用"/*"表示注释的开始,以"*/"表示注释的结束;跟C程序中一样,在C++中这种类型的注释不能嵌。

你在你的C++程序中加注释的第二个方法是采用单行注释法。单行注释以"//"作为注释的开始,结束则在行尾。除了结构上的行尾外(也就是换行与回车的结合),单行注释再也没有特别的注释终止符。

典型地,C++程序员使用类似于C的注释方法来进行多行注释,而把C++风格的单行注释留给短注释使用。

举例

1. 下面的程序包含C与C++风格的注释:

```
/*
   This is a C-like comment.
   This program determines whether an integer
   is odd or even.
*/

#include <iostream.h>

main()
{
  int num;  // this is a C++, single-line comment

  // read the number
  cout << "Enter number to be tested: ";
  cin >> num;

  // see if even or odd
  if((num%2)==0) cout << "Number is even\n";
  else cout << "Number is odd\n";

  return 0;
}
```

2. 虽然C注释不能嵌套,但是在多行注释中可以嵌套单行C++注释。

例如,下面的语句十分有效:

```
/*
   This is a multiline comment
   inside of which // is nested a single-line comment.
   Here is the end of the multiline comment.
*/
```

对你来说,单行注释能够嵌套入多行注释的事实使得为了调试目的的若干行代码的解释变得容易。

练习

1. 作为一个练习,测试下面的注释(它在C++风格的单行注释中套入了类似于C的

注释)是否有效：

```
// This is a strange /* way to do a comment */
```

2. 请你给 1.1 节的练习的答案加上注释。

1.4 类：简单的一瞥

C++中最重要的概念可能就是类。类是用来生成对象的机理，因此，类也是许多C++特点的核心。虽然类的主题贯穿整本书，有十分详细的讲解，但是由于类对于C++编程是如此的基本，所以，有必要在这里作一个简要的概述。

类的声明语法类似于结构的语法。它的一般形式如下：

```
class class-name {
    private functions and variables of the class
public:
    public functions and variables of the class
} object-list;
```

在类的声明中，对象表是可选的；就象结构一样，你可以在此之后再声明类对象，如果必须的话。虽然类名技术上也是可选的，但从实践的角度来看它本质上总是必须有的，之所以会这样的原因是类名是一种用来声明类对象的新的类型名称。

在类声明内部声明的函数与变量被称为类的成员。按照约定，类内部的所有函数与变量都是那个类专有的。这就意味着：它们只可以被该类的其他成员访问。为了声明公共类成员，应当使用后面带冒号的关键字 Public。在 Public 后面声明的各个函数与变量既可以被类的其他成员访问，也可以被包含该类的程序的任何其他部分访问。

下面是简单的类声明：

```
class myclass {
    // private to myclass
    int a;
public:
    void set_a(int num);
    int get_a();
};
```

这个类有一个专有变量以及两个公共函数 Set-a()与 get-a()。注意，函数在类中是以它们的原型形式被声明的。被声明为类的一部分的函数叫做成员函数。

因为 a 是专有的，所以它不可以被 myclass 之外的任何代码访问。但是，因为 set_a()与 get_a()是 myclass 的成员，他们可以访问 a。而且，由于 get_a()与 set_a()被声明为公共成员，因此可被包含 myclass()的程序的任何其他部分调用。

虽然，函数 get_a()与 Set_a()被 myclass 声明，但是，它们还没有被定义。为了定义成员函数，你必须将成员函数所属类的类型名与函数名联按在一起。这一问题的处理方法是在函数名前面加上类名，而用两个冒号将两者隔开。这两个冒号被称为作用域分辨算子。例如，下面是函数成员 get_a()与 Set_a()被定义的方法：

```
void myclass::set_a(int num)
{
  a = num;
```

```
}
int myclass::get_a()
{
  return a;
}
```

注意,Set_a()与 get_a()能够访问 myclass 的专有成员 a。因为 Set_a()与 get_a()是 myclass 的成员,它们能够直接访问 myclass 的专有数据。

用下面的一般格式,可以定义成员函数:

```
type class-name::func-name(parameter-list)
{
  ...//body of function
}
```

myclass 的声明并没有定义任何类型为 myclass 的对象——它仅仅在一个将被生成的对象被实际声明时候,定义了该对象的类型。用类名作为类型说明符可以生成对象。例如,下面这行语句声明了两个类型为 myclass 的对象:

```
myclass ob1, ob2; //these are objects of type myclass
```

记住:类声明是一个定义新类型的逻辑抽象,该新类型可以测定该型的对象看起来象什么。对象声明可以建立该类型的物理实体(也就是:对象将占据内存空间,但是类型定义不占据)。

一旦类的对象生成了,你的程序就可以用与访问结构成员完全相同的方法——点(句号)算子去引用公共成员。假定完成了对象声明,下面的语句将为对象 ob1 与 ob2 调用 Set_a():

```
ob1.set_a(10); //sets ob1's version of a to 10
ob2.set_a(99); //sets ob2's version of a to 99
```

就象注释指出的那样,这些语句将 ob1 中的 a 的拷贝置为 10,ob2 中的 a 的拷贝置为 99。每一个对象都包含类中声明的全部数据的拷贝。这就意味着:ob1 中的 a 和 ob2 中的 a 是显然不同的。

记住:类的每一个对象都有它自己的在类中声明的每一个变量的拷贝。

举例

1. 作为一个简单的例子,这个程序用文中描述的 myclass 来设置 ob1 与 ob2 的 a 值,并显示 ob1 与 bo2 的 a 值:

```
#include <iostream.h>

class myclass {
  // private to myclass
  int a;
public:
  void set_a(int num);
  int get_a();
};

void myclass::set_a(int num)
{
  a = num;
```

```
}

int myclass::get_a()
{
  return a;
}

main()

{
  myclass ob1, ob2;

  ob1.set_a(10);
  ob2.set_a(99);

  cout << ob1.get_a() << "\n";
  cout << ob2.get_a() << "\n";
  return 0;
}
```

你应当预料到,这个程序在屏幕上显示的是数值 10 与 99。

2. 在前面例子的 myclass 中,a 是专有的。这意味着只有 myclass 的成员函数可以直接访问 a(这是要求有公共函数 get—a()的原因之一)。如果你试图用你程序中那些并不是类成员的部分访问类的专有成员,将会发生编译时间错误。例如,假定 myclass 被定义为前面范例中的形式,下面的 main()函数会导致错误:

```
// This fragment contains an error.
#include <iostream.h>

main()
{
  myclass ob1, ob2;

  ob1.a = 10; // ERROR! cannot access private member
  ob2.a = 99; // by non-member functions.

  cout << ob1.get_a() << "\n";
  cout << ob2.get_a() << "\n";

  return 0;
}
```

3. 正如可以有公共成员函数一样,也可以有公共成员变量。例如,如果 a 在 myclass 的公共部分被声明,那么 a 就可以被程序的任意部分引用,如下所示:

```
#include <iostream.h>

class myclass {
public:
  // now a is public

  int a;
  // and there is no need for set_a() or get_a()
};
main()
{
  myclass ob1, ob2;
```

```
  // here, a is accessed directly
  ob1.a = 10;
  ob2.a = 99;

  cout << ob1.a << "\n";
  cout << ob2.a << "\n";

  return 0;
}
```

在这个例子中，因为 a 是作为 myclass 的公共成员被声明的，所以可以直接被 main()访问。注意一下点算子是如何被用来访问 a 的。一般来说，无论你是正在调用成员函数，还是访问成员变量，总会要求对象名后跟点算子，点算子后跟成员名，以此来指定你引用了哪一个对象成员：

4. 为了尝尝对象的力量，让我们看一个更实际的例子。这个程序建立了被称为 Stack 的类，以用来实现存放字符的堆栈：

```
#include <iostream.h>

#define SIZE 10

// Declare a stack class for characters
class stack {
  char stck[SIZE]; // holds the stack
  int tos; // index of top-of-stack

public:
  void init(); // initialize stack
  void push(char ch); // push character on stack
  char pop(); // pop character from stack
};

// Initialize the stack
void stack::init()
{
  tos = 0;
}
// Push a character.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Pop a character.
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
```

```
}
main()
{
  stack s1, s2;  // create two stacks
  int i;
  // initialize the stacks
  s1.init();
  s2.init();

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  return 0;
}
```

这个程序显示下面的输出：

```
Pop s1: c
Pop s1: b
Pop s1: a
Pop s2: z
Pop s2: y
Pop s2: x
```

现在让我们仔细分析一下这个程序。类 Stack 包含了两个专有变量：Stck 与 tos。数组 Stck 实际上保存压入堆栈的字符，tos 包含堆栈顶部的索引。公共堆栈函数是 init()、push()、与 pop()，它们分别初始化堆栈，将一个数压入栈中以及将一个数弹出堆栈。

在 main()中，生成了两个堆栈 S1 与 S2，并且三个字符都被压入每个栈中。理解堆栈对象与堆栈对象之间是分离的这一点很重要。也就是说：压入 S1 中的字符无论如何不会影响压入 S2 中的字符。每一个对象都包含它自身的 Stck 与 tos 的拷贝。这个概念是理解对象的根本。虽然，所有类对象都分享它们的成员函数，但是，每一个对象都建立并保持自己的数据。

练习

1. 如果你还没有这么做的话，那么请输入并运行这一节的范例中的程序。

2. 建立一个被称为 card 的类，用来保存图书馆卡片分类记录。用这个类存储书的题目、作者，馆中拥有的数量。其中题目与作者以字符串的形式保存，书的数量为整数。用被称为 store()的公共成员函数存储书的信息，用公共成员函数 show()显示这些信息。用一个较短的 main()函数来演示这个类。

3. 建立一个保存环形整数队列的队列类。使队列为 100 位整数长。用一个较短的 main()函数来演示队列的运算。

1.5 C与C++的不同之处

虽然C++是C的超集，但是它们之间仍有一些细微的不同，这必然会在你开始编写C++程序的时候影响你。虽然每点不同之处都很微小，但它们仍然渗透到C++程序中。因此，在继续进行之前，花点时间重温一下刚才提到的那些不同之处。

首先，在C中，当函数没有参数的时候，它们的原型会有一个字void在其函数参数表中。例如，在C中，如果函数f1()没有参数(并且返回char)，那么，它的原型就会如下所示：

```
char f1(void);
```

但是，在C++中，void是可选的。因此，f1()的原型通常写成下列形式：

```
char f1();
```

在这一点上，C++不同于C的地方在于它指定了一个空的参数表。如果上面的原型发生在C程序中，仅仅意味着没有关于函数的参数的内容。但是在C++中，则意味着函数没有参数。这就是前面的几个例子没有用void明确声明空参数表的原因(用void声明空参数表是非法的；它太冗长。因为大多数C++程序员用一种近乎宗教的热忱追求效率，所以你也许永远也看不到void在这种方式中的运用)。记住：在C++中这两个声明是等价的：

```
int f1();
int f1(void);
```

C与C++的另一个微妙不同之处在于：在C++程序中，所有的函数都必须是原型化的。记住，在C中，虽然推荐使用原型，但从技术的角度看却是可选的。然而，在C++中，原型却是必需的。就象前一部分的例子所表示的那样，类所包含的成员函数的原型也用作它的一般原型，而不一定专门要求另一个单独的原型。

C与C++的第三个不同点是：在C++中，如果函数被声明为具有返回值，那么它一定要返回一定的值。也就是说，如果函数有一个返回型，且不是void型，那么，该函数内的任何return语句都必须包含一个值。实际上在C中，任何一个非void函数都不要求返回值。如果没有返回值，就会“返回”一个垃圾值。

C与C++的另一个你经常会遇到的差别存在于声明局部变量的地方。在C中，局部变量优先于任何“action”(动作)语句，只能在程序块的开始处声明。在C++中，局部变量则可以在任何地方声明。这种方法的一个优点是局部变量可以在离它们被使用的地方最近的部位声明，于是，有助于防犯局部变量的负效应。

举例

1. 在C程序中，如果没有命令行变元的话，声明main的一般形式如下所示：

```
main(void)
```

但是，在C++中，void的使用是冗余的，不必要的。

2. 这个简短的C++程序不可能编译，因为函数sum()没有被原型化。

```
// This program will not compile.
#include <iostream.h>
```

```
main()
{
  int a, b, c;
  cout << "Enter two numbers: ";
  cin >> a >> b;
  c = sum(a, b);
  cout << "Sum is: " << c;

  return 0;
}

// This function needs a prototype.
sum(int a, int b)
{
  return a+b;
}
```

3. 下面的程序表示局部变量如何在块的任一部位被声明。

```
#include <iostream.h>

main()
{
  int i; // local vars declared at start of block

  cout << "Enter number: ";
  cin >> i;

  // compute factorial
  int j, fact=1; // vars declared after action statements

  for(j=i; j>=1; j--) fact = fact * j;

  cout << "Factorial is " << fact;

  return 0;
}
```

在这个简短的例子中,j 与 fact 在它们的第一个使用点附近被声明几乎没有什么价值。但是,在大函数中,靠近第一个使用点进行变量声明的能力有助于你简化你的代码,并防止不希望出现的效应。在涉及较大的函数时,C++的这个特点在编程中是经常使用的。

练习

1. 下面的程序不会被作为C++程序编译。为什么?

```
#include <iostream.h>

main()
{
  char s[80];

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

  cout << "Length is: ";
  cout << strlen(s);

  return 0;
}
```

2. 请你试编一个C++程序，使局部变量在不同的位置被声明。编一个C程序，作同样的工作，注意一下哪个声明会发生错误。

1.6 介绍函数重载

类之后，下一个最重要、最具普遍性的C++特点可能是函数重载。函数重载不仅可以提供一种机理，通过该机理，C++可以获得一种多态，而且它还可以构成C++编程环境能动态扩展的基础。因为重载的重要性，下面作一个简洁的介绍。

在C++中，只要函数变元的类型不同，或者函数参数的数量不同——或者二者兼备，两个或者两个以上的函数就可以分享相同的名称。当两个或者两个以上的函数分享同样名称时，它们被叫做重载。通过用相同的名称来引用相关联的多种操作，重载函数可以减少程序的复杂性。

重载一个函数十分容易：只要声明并定义所有要求的版本，编译器会根据调用该函数时所用参数的数目和/或类型自动选择正确的版本来调用。

注意：在C++中，也可以重载算子，不过，在你能够理解算子重载之前，你必须了解更多的有关C++的知识。

举例

1. 重载的一个主要用法就是获得编译时间多态，多态体现了一个接口、多种方法的哲学，就象你知道的那样，在C语言编程中，有许多仅通过它操作数据类型来区分的相关函数是很普通的。这种情况的典型范例可以在C标准库中找到。正如本章前面描述的那样，库中包含了函数abs()、labs()与fabs()，它们分别返回整数，长整数与浮点数的绝对值。

但是，因为是三种不同的数据类型，就必须有三个不同名称的函数，所以情况就比实际的情况更为复杂。针对三种情况，绝对值被返回，唯一不同的是数据类型。在C++中，你可以通过对三种数据类型重载一个名称来校正这种状况，如下面的例子所示：

```
#include <iostream.h>

// Overload abs() three ways
int abs(int n);
long abs(long n);
double abs(double n);
main()
{
  cout << "Absolute value of -10: " << abs(-10) << "\n";
  cout << "Absolute value of -10L: " << abs(-10L) << "\n";
  cout << "Absolute value of -10.01: " << abs(-10.01) << "\n";

  return 0;
}

// abs() for ints
int abs(int n)
{
  cout << "In integer abs()\n";
  return n<0 ? -n : n;
}
```