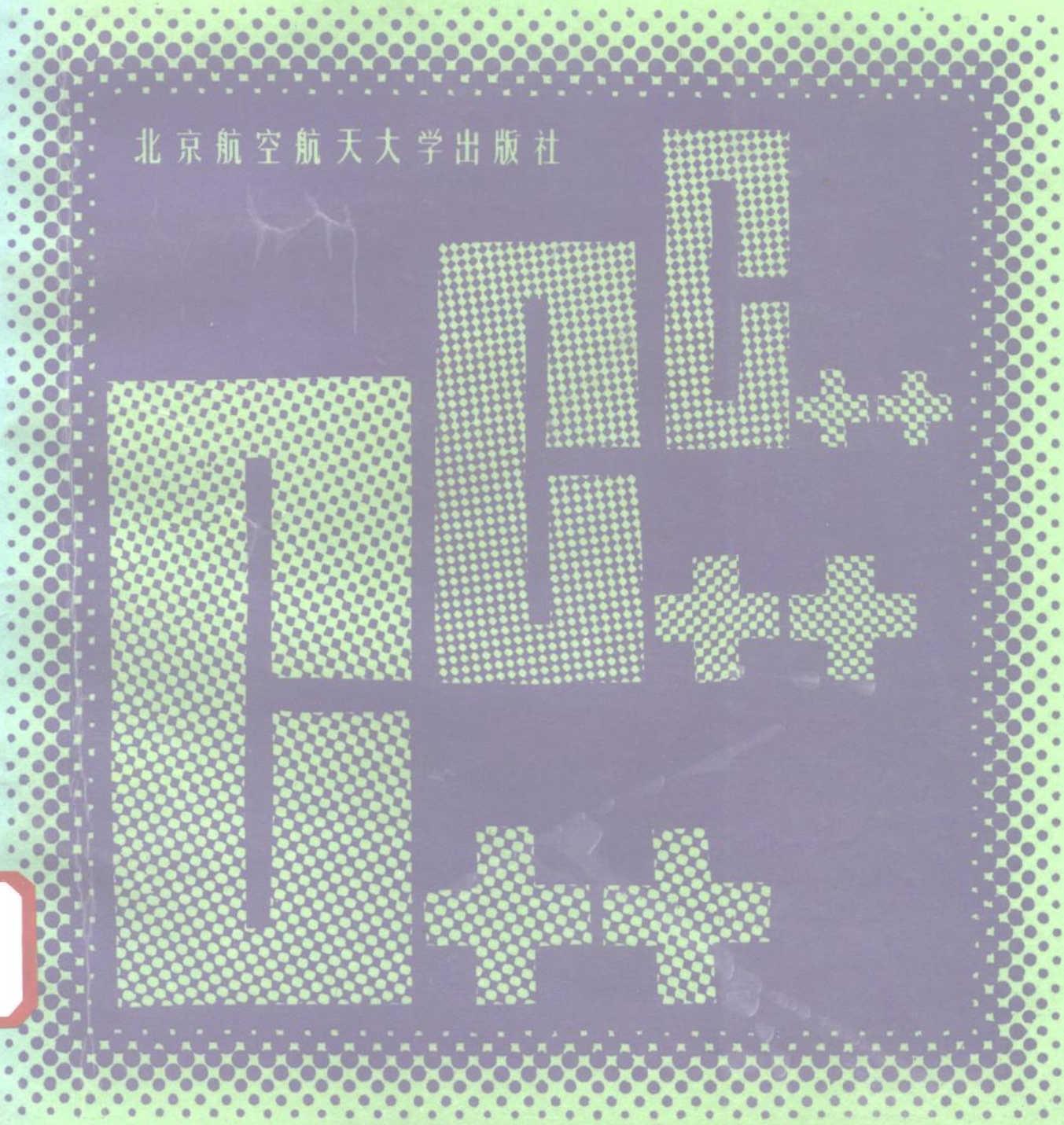
C++
程序设计语言教程
（编程技术）

麦中凡

北京航空航天大学出版社



C++程序设计语言教程（编程技术）
北京航空航天

TP312

C++程序设计语言教程

（编程技术）

麦中凡　严建新　刘书舟　著

北京航空航天大学出版社

内 容 简 介

本书为《C++程序设计语言教程(语言基础)》的姊妹篇。两书内容既有联系,又相对独立。两书合一是一套完整的教材。主要内容包括:数据抽象技术,单继承和多继承编程应用,虚函数和多态性,面向对象编程(OOP)方法、步骤及完整的示例,C++的代码重用机制,类库设计技术,C++与C语言和80X86汇编语言的共用等等。在学习本书之前,读者应掌握C++程序设计语言基础的内容。

本书可以作为大专院校C++高级程序设计的教材,对涉及软件工程和面向对象程序设计课程的大专院校的教师和学生来说,也是一本理想的参考书。

图书在版编目(CIP)数据

C++程序设计语言教程:编程技术/麦中凡等编著.—
北京:北京航空航天大学出版社,1996.4
ISBN 7-81012-642-3

Ⅰ.C… Ⅱ.麦… Ⅲ.C语言-程序设计 Ⅳ.TP312C

中国版本图书馆CIP数据核字(96)第02913号

C++程序设计语言教程(编程技术)

C++ CHENGXU SHEJI YUYAN JIAOCHENG (BIANCHENG JISHU)

麦中凡 严建新 刘书舟 著

责任编辑 陶金福

责任校对 李宝田

北京航空航天大学出版社出版

北京学院路37号(100083) 2015720(发行科电话)

新华书店总店北京发行所发行 各地书店经销

通县觅子店印刷厂印装

*

787×1092 1/16 印张:16.25 字数:414千字

1996年5月第一版 1996年5月第一次印刷 印数:5000册

ISBN 7-81012-642-3/TP·209 定价:22.00元

前　言

本书为《C++程序设计语言教程(语言基础)》的姊妹篇。两书内容既有联系，又相对独立。两书合一是一套完整教材。

《C++程序设计语言教程(语言基础)》介绍了C++的语法和面向对象编程(OOP)的基本概念，以使读者熟悉语法，编制简单面向对象程序为目的。在此基础之上，本书介绍C++编程技术。全书共9章，第一章分别从结构化编程、数据抽象和面向对象编程三个角度对C++进行"解剖"和分析；第二章介绍数据抽象技术；第三章和第五章分别是单继承和多继承编程及应用；第四章介绍虚函数应用，强调了面向对象的多态性编程和非多态性编程的对比；第六章讲解OOP的详细步骤并给出了一个完整的示例；第七章讨论对象和重用，并介绍C++的四种代码重用技术；第八章介绍类库设计；为满足实际工程需要，在本书最后，即第九章，对C++和C语言、80X86汇编语言的混合编程技术作了较为详细的讲解。

本书力求重点突出，通过C++语言语法结合实际应用和示例更深入的解释C++编程技术，同时兼顾我们对面向对象研究的一些见解和体会，着眼点完全在于希望读者通过本书学会用C++组织OOP程序并能顺利实现，但本书并不是面向对象开发(OOD)和OO软件工程的教程，故仅限于面向对象模型已清楚的OOP程序设计。

北航计算机系陈羽中同学参与调试了书中部分程序，特致谢意。

全书不当之处，敬请有识之士指正。

编　者
于北京航空航天大学计算机系
1996年1月

目　　录

此为试读，需要完整PDF请访问：www.ertongbook.com

第 1 章

C++：多范型程序设计语言

学完《C++程序设计语言教程(语言基础)》(作为第一卷),初步熟悉了C++编程以后,自然希望在编程技术上有所提高,本书(作为"教程"的第二卷)即为此而写。本书将通过大量实例的设计实现来说明如何综合利用C++各个语法特征完成问题求解;从而使读者加深对C++程序设计语言的认识,知道它能做什么?在什么情况下应怎么做?

本章先讨论C++程序设计语言支持哪些程序类型。C++语言是一种支持多范型程序设计的语言。在1.1节中简要讨论各种编程范型以后,将在本章的其它小节中详细介绍C++对各种不同编程风范的支持。其中,1.2节讨论C++对传统编程风范的支持,并对C++对C的非面向对象增强进行了总结,为从C语言过渡到C++的程序员提供了有力的技术支持;1.3节和1.4节则将分别具体分析介绍C++对数据抽象和面向对象编程的支持。

1.1 程序设计范型

范型(paradigm)是做一件事的规范做法,包括想法(构成的模型)、做法(制作的规范、风格)和结果(实例形状)。程序设计总是要按照某种计算模型,以某种开发过程开发出某种形式的程序。称之为程序设计范型。例如,以算法过程施加于数据结构并构成相互共享或独立的模块程序,无论是自顶向下逐步求精,还是由底向上聚合成型,都叫过程式程序设计范型。它最主要的特征是算法过程和过程间的调用,简单地说,程序=过程+调用。

再如,面向对象程序设计范型,把一切计算对象都看作是封装了的对象在接受外界触发之后对象内部计算而导致对象状态的改变。对象相互之间发消息即触发,所以,面向对象程序=对象+消息。

此外,逻辑式程序以一阶谓词描述事实、规则、目标,按规则匹配事实以证实目标断言的真伪。如何组织控制事实和规则(知识库)以求正确的逻辑匹配成为主要特征。因此,逻辑程序=逻辑+控制。

当然,每种范型也有其更细致的特点,例如,过程式范型的每一计算过程必须以严格的结构化形式表达,即只能是顺序语句、条件分支、循环迭代及其嵌套构成计算过程,故称为过程式结构化程序设计范型。程序设计方法学可能规定得更细致一些,例如,Yourdon的结构化程序设计的SA-SD方法学给出的范型是自顶向下逐步求精的过程式、结构化的程序设计范型。Jackson的JSP方程学给出的范型则是面向数据结构,由底向上,过程式、结构化程序设计范型。

一般说来,程序的表达是程序设计范型的重要侧面。程序设计语言往往为表达某种范型而设计。例如,过程范型是有控制的赋值过程(即控制一段连续的赋值语句以改变程序状态)。没

有赋值语句不能叫过程式。例如，函数式语言只能按函数求值，不能赋值。逻辑式语言也没有赋值语句，只能匹配求值。所以有过程式、函数式、逻辑式、对象式、数据流式、关系式程序设计语言之分。这种程序设计语言叫单范型语言。例如，Pascal，ML，Prolog，Sonalltake，VAL，SQL（显然这是就较大范畴讨论的范型，在较小范畴中，如结构化和非结构化也是两种不同范型，FORTRAN-77 两者都支持。而 FORTRAN IV 只支持后者。这里不说 FORTRAN-77 是多范型程序设计语言）都是单范型语言。如果一种程序设计语言支持多种程序设计范型，则称多范型语言，例如，C++，Add-95（过程式、面向对象式），CLOS（函数式、面向对象式）。也就是说，按两种范型及其混合形式编写的程序，编译（或解释）系统均可处理。

在第一卷绪论中已论述 C++是反映当今软件技术的程序语言，现在从程序设计技术的角度讨论程序设计范型的一些发展，而这些技术 C++都能支持。

1.1.1 结构化程序设计范型

第三代程序设计语言如 Pascal，C，Ada-83，FORTRAN-77 都是支持结构化程序设计的。结构化程序消除了程序控制的混乱。反映到开发过程是人们只能把算法组织在三种典型结构（顺序赋值、条件分支、迭代）之中。严格保持控制结构的完整，只能按一个出口一个入口的程序块层层嵌套，不能从半块中插入。过程控制调用实质是嵌套的显式写法。由于嵌套，数据结构则有全局、局部共享和完全局部（私有）之分。过程和函数是十分重要且唯一的程序结构形式，对应的逻辑是完成某种程序的功能（做某件事，算某些数）。

结构化程序设计在某种程度上限制程序员的思想，但能换取大型程序的安全，所以第二代语言均结构化。此处不再赘述。C++是 C 语言的超集。在形成 C++时将 C 语言原有的缺陷作了改进，也就是把 C++当作 C 使用，编制结构化程序，比 C 语言更好。

1.1.2 数据抽象程序设计范型

数据抽象是 20 世纪 70 年代末发展起来的一种程序设计范型。它以抽象数据串型表达复杂的程序实体，这样，可以使程序计算局部化，简化了控制。首先解释什么是抽象数据类型。

程序设计语言系统一般只提供基本类型和简单的结构类型（数组、记录、变体记录、联合、文件）。第三代程序设计语言之后，用户可以以这些基本类型定义复杂数据结构的类型，例如，一个记录中成分是数组，数组元素又是记录……但我们知道，类型定义了一个数据值集和可以施加到这个数据值集上的操作集，例如，整数类型的值集是－maxint 到 maxint 范围内的整数，以及整数加、减、乘、除和存数取数这些操作。第三代语言只许程序员有充分扩充复杂数据结构的自由，而无法扩充操作集。例如，C 语言无法定义一个堆栈类型，只能写出堆栈处理程序：

```
struct stack{
    char * p;        //堆栈指针
    char * v;        //堆栈
    int size;        //堆栈大小
}
BOOL is_full(stack * sta)
{
      return (sta->v - sta->p) < sta->size :FALSE ? TRUE;
```

```
}
BOOL is_empty(stack * sta)
{
      return (sta->v - sta->p) <= 0 :TRUE ? FALSE;
}

void push (stack * sta,char c)
{
      if(! is_full(sta))
            *(sta->v ++)=c;
}

char pop()
{
      if(! is_empty(sta))
            return *(--sta->v);
      else
            return NULL;
}
```

其中，pash(压入)、pop(弹出)、is－full(已满)、is－empty(已空)都是施加于堆栈体上的操作，从而构成数据堆栈所具有的行为。但C(或Pascal)无法定义堆栈变量，stack类型变量只不过是堆栈体数据结构的变量。虽然程序员注意地按堆栈使用它也可以实施堆栈的功能，但无法防止本文件内其它函数操作此堆栈体(有可能是错误指令导致的转换操作)。

抽象数据类型把若干个用户定义的数据类型及若干个施加于这些类型的操作用一个更高层的类型名封装起来。该类型则叫抽象数据类型(Abstruct Data Type，简称ADT)。抽象数据类型经过封装之后形成了一个对外的界面。外界只需按操作的名称，给定所需的参数即可调用此操作使本类型刻画的值集有所改变，无需考虑操作实现的细节。这样，定义在各操作内部的数据也作为实现细节外界不可见，这就是所谓的数据隐藏。数据隐藏还表现在有些数据和操作可以不出现在界面上。下例是is-full和is-empty作为内部操作的ADT堆栈程序：

```
class stack{
private:
      char * p;
      char * v;
      int      size;

      BOOL is_full();
      BOOL is_empty();

public:
      void push (char c);
      char pop();
```


此为试读，需要完整PDF请访问：www.ertongbook.com

```
}

BOOL stack::is_full()
{
        return (v - p) < size ;FALSE ? TRUE;
}

BOOL stack::is_empty()
{
        return (v - p) <= 0 ;TRUE ? FALSE;
}

void stack::push (char c)
{
        if(! is_full())
              * v ++=c;
}

char stack::pop()
{
        if(! is_empty())
              return *(--v);
        else
              return NULL;
}
```

读者也许注意到，定义在界面上的函数只写了函数的型构(Signature)，没有写函数体。而函数体写在同一文件的抽象数据类型定义之外，并冠以 ADT 名字和作用域分辨符::。这是抽象数据类型的另一个重要特征：数据和操作的规格和实现它们的细节可以显式的分开，即先定义 ADT 的规格(即外界用户可见的界面)指明本 ADT 能“做什么”，然后再定义它们“怎么做”。这不仅仅是为了界面干净清晰，更重要的是抽象数据类型使程序更易维护。例如，某个程序操作实现有了修改，只要该操作的型构(即规格说明)不变，那么 ADT 界面定义也不变，用到该 ADT 的其它程序部分也无需改变。

封装本身就是模块化。正因为如此，ADT 程序设计范型首先要描述比函数(或过程)要高层的大模块(ADT 定义)，因为它们自己已经有了自己的数据和操作，则主控程序变得极其简单：给定初始数据建立各 ADT 的实例(变量)；发出调用各 ADT 实例作各自的计算，最后得出结果。在时间上，这些封装的 ADT 模块可以事先编写，编译后作为程序资源放在库中。只要写个很短的主控程序，编译后和要用的资源连接即可运行一个很大的程序。这就是 ADT 支持软件重用，一般要求分别编译的原因。其程序结构如图 1.1 所示。

综上所述，抽象数据类型是用户定义的一组数据类型，以及施加于该组类型上的一组操作，并以 ADT 名字封装。抽象数据类型的型构和其全部实现在程序正文上有显式分离。抽象数据类型极易实现数据隐藏并支持易维护性。要求分别编译以支持软件重用。

利用抽象数据类型设计程序即称之为数据抽象。

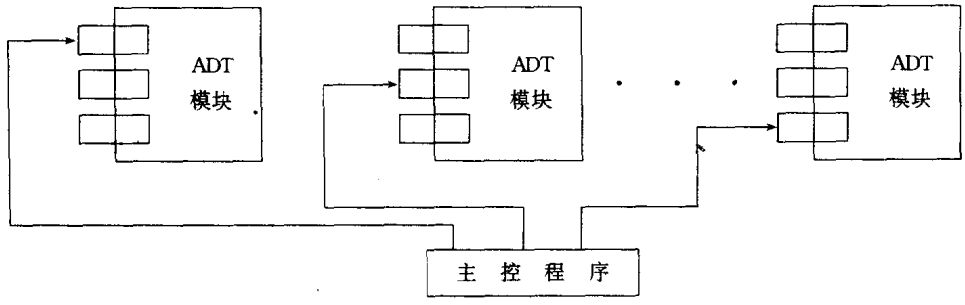


图 1.1 ADT 程序结构

1.1.3 面向对象程序设计范型

程序对象并非新鲜概念。程序语言中所有运算实体都是程序对象，如常量、变量、函数等，只要它有名字、值和容许的操作。例如，整常数 123，整变量 a，它们都隐含有容许的整数四则操作，不可开方。抽象数据类型的变量 s：

```
stack s (int size);
```

名为 s，值为 size 个整数值，容许 stack 中显式定义的操作。所以它也是对象。面向对象中把抽象数据类型也叫做对象。为了区别只给说明不参与运行的类型称类对应，变量则对应称为实例对象。类对象参与运行，运行的目的是生成或撤销实例对象，并赋以初值。实例对象各封装数据值（一般运行时发生），类对象名封装这些值的类型和容许的操作（编写程序时给出，静态发生）。类是实例的样板的制造厂。

面向对象和抽象数据类型在类和实例（类型和变量）方面几乎一样。但在支持重用方面走得更远。面向对象重要的特征是有继承，即库中放的可重用类对象不再是如 ADT 那样的平面模块（如图 1.1 所示），而是有语义相关的类层次树（单继承）或网（多继承）。类库愈是积累，树（或网）越大。每次为了一个新应用派生几个类，直接重用几个类，重新设计很少的类即可完成程序设计。到了一定时候，只需按应用分析设定一组初值向库中的某些类发消息，不作任何程序设计即可运行程序。其示意图见图 1.2。

图中假定 B,C 是类库中已有的类，D,E 是新设计的派生类，A 是基体没有继承全新设计的类（直属类库树根 Object）。再设计一个主控对象 S 向 A,B,C,D,E 五个类对象发消息（带有生成实例对象初值），于是得到 a1,a2,b1,b2,b3,c1,c2,d,e1,e2,e3,d4 等 12 个实例对象；它们又相互发消息，各对象接受消息后按样板规定的操作改变自己的数据。运行后所有程序对象的最终状态即为结果。如果对象是打印机的实例对象，其 它对象把结果值以消息传递给它，则可全部打印出来。

所以，面向对象程序设计的范型是选用、派生、新设计程序中需要的类对象。熟悉类库是重要的，核心是对象分析技术，并充分表达客观世界问题对象中原有的动态和开发性。

图 1.2 中的示意图适合于 C++风格，有一主控对象 S。如果 S 退化与只触发 A，A 除了生成实例外又触发 B,C,D 生成实例，向它们发消息；C 又触发 E …。这样就成了无主控对象风格的（如 Smalltalk）。此外，实例对象动态先后生成，当然也可以先后死亡。程序运行终结时也可以只有一个实例对象，它是高度动态的。

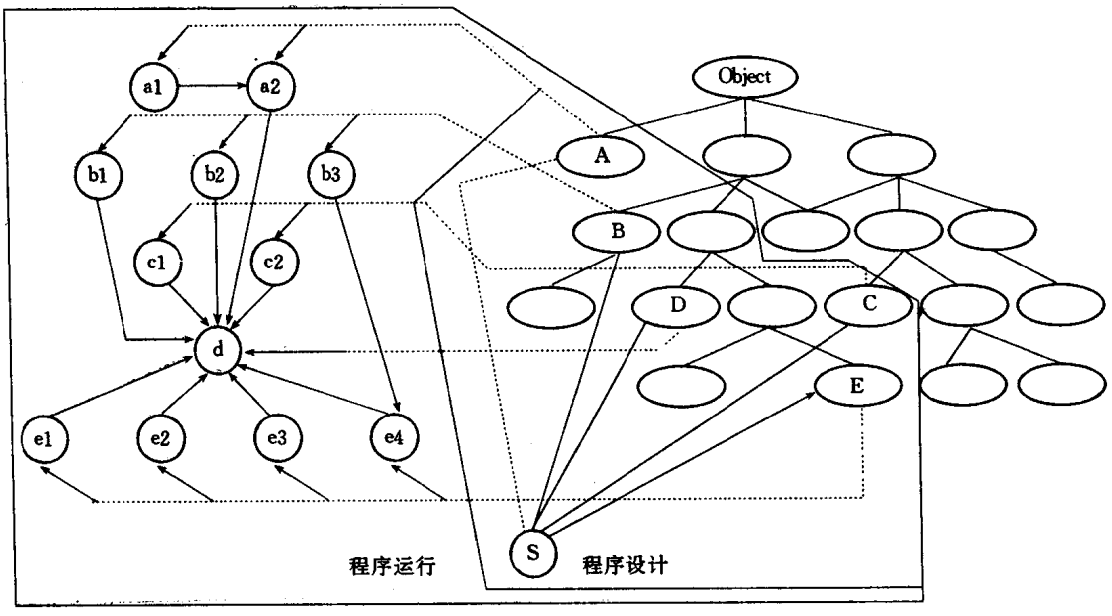


图 1.2　面向对象范型示意图

1.1.4　数据抽象与面向对象编程之比较

基于抽象的自定义数据类型实际上类似于以抽象的层次定义了一些“黑盒”，一旦被定义，它们就不再与程序的其他部分进行任何真正意义上的交互，除了直接改写源码，对其定义进行修改以外，基本上没有其他方法能够使其适应于新的应用。这样就可能导致缺乏灵活性。比如说，要在一个图形系统中定义 shape 类来描述图形形状，假定系统必须支持圆、三角形、正方形。在定义好两个基本类型：

```
class point {/ * … * /}
class color {/ * … * /}
```

以后，可以这样来定义 shape：

```
enum kind {cirle,triangle,square};
class shape {
   point center;
   cdor col;
   kind k;    //形状描述
public:
     point where() {return center;}
     void move (point to) {center=to;draw(),}
     void draw();
     void rotate(int);
       // 其他操作};
```

其中的变量 k 是必须的，诸如 draw()，rotate() 等操作函数根据它的值来确定它们正在处理的形体。例如，函数 draw()可以这样来定义：

```
void shape::draw()
{
switch (k) {
   case cirde:
        // 画圆
          break;
   case triangle:
        // 画三角形
          break;
   case square:
        // 画正方形
        break;
    }
}
```

粗看起来，这些定义都是理所当然的，没有什么不好。请考虑一下，上述 draw( )函数是在什么情况下面写出来的呢？答案很明显：在知道了将处理的所有形状以后。系统每多处理一种形体，这些操作函数的 case 语句代码会随之增长。定义新形体时，必须对每个操作函数进行检查并作必需的修改，不直接与每个函数的源码打交道的话，是不可能在系统中增加其处理的形状的。而这种代码扩充不仅需要技巧，还容易给原有的代码引入错误。此外，shape 类型以这 种方式定义，实际上限定了每种形体的表示框架，对于特殊形体的加入来说，这种固定的框框不免使程序员有画地为牢的局促感。

数据抽象造成这种缺陷，其主要原因在于，数据抽象进行时并不强调形体的通用属性(如有颜色，可以被 画出来，等等)和特定形体的具体属性(如圆有半径，必须根据其半径和圆心用弧画出来，等等)之间的差别，它只强调抽象出所有能抽象出的属性。而面向对象编程则能表达出这种差别，并能很好地加以利用。

对于上述具体问题(shape 类型)，面向对象编程中的继承机制提供了一个很好的解决途径。首先，可以定义一个描述所有形体通用属性的类：

```
class shape {
      point center;
      color col;
      //…
public:
        point where ( ) { return center;}
        void move (point to ) {center=to;draw();}
        virtual void draw();
        virtual void rotate(int);
        //…
};
```

在这里，只定义操作函数的调用接口。定义特定形体时，必须指定它首先是 shape 的一种，然后才可指定它的具体属性，如：

```
class circle:public shape{
        int radius;
public:
        void draw() {/*…*/}
        void rotate(int) { } //此函数为空,圆无须旋转
};
```

这里的 shape 和 circle 仍然都是基于数据抽象的自定义类型，在 C++中，用“类”这个术语来描述它们。基类 shape 及其派生类 circle 通过继承机制，实际上就刻划了一对普遍与特殊的关系。

不需要刻划公有属性时，数据抽象就够用了。而利用继承和虚函数机制，来刻划类型之间的公有属性，从而表达出实体对象间的关系，是面向对象编程在实践中切实可行的必要保证。实际上，类型之间能被抽象出的公有属性的多少，决定了面向对象编程风范的可用程度。如在交互式图形系统等领域，面向对象编程大有用武之地；而在传统的数学计算等领域，抽象出各类型的公有属性难免有勉为其难之性质，在这种场合，面向对象编程并不会比传统的编程风范有更大的效用，大可不必费尽力气，而只得到了一个“已验证了面向对象技术的普遍性”之类的结果。

到这里，已经简单地了解了结构化编程、数据抽象和面向对象编程风范。下面将从常规编程风范、数据抽象和面向对象编程这三个角度来分别分析 C++ 语言的有关语言机制。

1.2 C++：常规编程中更好的 C

语言中对常规的结构化编程的支持，不外乎提供函数调用、算术运算、分支和循环等控制结构、I/O 处理等功能。C++从 C 中继承了这些基本的语言机制，并提供了一个专用库来处理 I/O。此外，和 C 语言中一样，C++通过独立编译机制支持模块化概念中的数据隐藏思想。

作为 C 语言的超集，C++继承了 C 语言的优点，概括而言，它对 C 语言从两个方面作了很多扩充，即面向对象增强和非面向对象增强。在 C++环境中编写程序，即使根本不用到它的数据抽象和面向对象机制，其功能也比常规 C 语言更强，编起程序来更灵活、更好用。这些优点在很大程度上都来源于 C++对 C 的非面向对象增强。从常规编程的角度，可以说 C++是一种“更好的 C”。在本节中，将重点总结 C++对 C 语言的非面向对象增强，这些增强虽然很小，却很重要。

1. 注 释

C++提供两种形式的注释符，除了支持 C 语言风格的注释符/*…*/外，它还支持新的注释符//(双斜杠)。/*…*/便于应用于大块的注释文字，其中的行数不限；而//适用于单行注释，自//符号起至该行结束处为注释内容，编译器不作任何处理。这两种注释符可以根据个人的爱好随意使用，在同一文本中可以同时出现。实际上，在前面的示例程序行中，已用到了这两种注释符。本书中的示例将完全根据具体情况选用/*…*/或//…。

2. 函数原型

函数原型现在已经是 ANSI C 标准中的一部分，故严格地讲，它现在可以不算是 C++对 C 的增强了。仍将它放在这里讨论主要有两个原因：一、由于其重要性；二、实际上，函数原型

确实是C++最早提出来的，尔后才被引入ANSI C。

函数原型通常如下定义：

Return Type FunctionName (type_1 parameter1,…,typeN parameterN)；但它的引入并不是为了装点门面，使用C++之前(也可以说ANSI C之前)的风格，编译器并不对函数进行类型检查，如果某一函数调用中用错了参数类型，或参数的顺序、数目有误，编译时这些错误都无法检测出来。C++和ANSI C中使用函数原型，上述这些错误编译器能轻易地检查出来，而不象以往，要到程序运行出现故障时，才使程序员意识到源代码中还有隐藏的错误。

3. 作用域限定符

作用域限定符::在C++中，可以用于几种情况，它既可用于类声明之外的成员函数的定义，也可用于改变变量的存取作用域。请看下面的示例程序：

```
//scope.cpp
#include <stdio.h>

float r=2.6;

int increment (int k)
{
    int r=k+15;

    printf("In function increment:the value of r within increment=%d\n",r);
    printf("In function increment:the value of r outside increment=%0.2f\n\n",::r);
    return r;
}

void main()
{
   int r=increment(20);

   printf ("In main:the value of r=%d\n",r);
   printf ("In main:the value of outside main =% 0.2f\n\n",::r);
}
```

程序中，函数increment和mian中定义的自动变量r屏蔽了全程变量r，而通过使用作用域限定符，可以在increment和main中访问全程变量r。程序的运行结果如下所示：

```
In function increment:the value of r within increment=35
In function increment:the value of r outside increment=2.60

In main:the value of r=35
In main:the value of outside main=2.60
```

4. 灵活的声明

C语言要求在某个作用域内的所有声明要C++在此作用域的开始处给出，或是简单地规定：全程变量声明必须在任何函数之前出现，而局部变量则必须放在函数中的任何可执行语

此为试读，需要完整PDF请访问：www.ertongbook.com