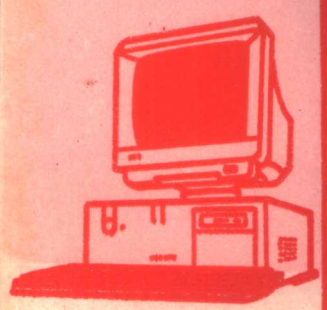
C++语言和面向对象有限元程序设计

孔祥安　编著

西南交通大学出版社



C++语言和
面向对象有限元程序设计

孔祥安　编著

西南交通大学出版社

内 容 提 要

程序设计方法的革命性变革可归纳为两个O (Object Oriented，面向对象程序设计) 和两个P (Parallel Programming，并行设计)。其中并行设计和计算在有限元等数值计算领域中的应用，已有大量的成果。面向对象程序设计方法在有限元等方面的工作，国内基本上是空白。

本书旨在介绍面向对象程序设计这一软件工程学中具有重大意义的最新发展及其在有限元、边界元程序设计中的应用前景，其中包括作者的工作。

全书共分八章。前两章介绍面向对象程序设计的基本方法和C++语言的要点。后六章介绍面向对象有限元的系统分析方法以及一个建立在C++类的继承层次基础上的面向对象有限元软件包。

本书可供高等院校力学和工程类专业、计算机软件专业的师生和软件设计人员参考。

C++语言和面向对象有限元程序设计

孔祥安 编著

*

西南交通大学出版社出版发行

(成都 九里堤)

新华书店经销

郫县印刷厂印刷

*

开本：787×1092 1/16 印张：18.625

字数：430千字 印数：500册

1995年8月第1版 1995年8月第1次印刷

ISBN 7—81022—819—6/T·145

定价：17.00元

前　言

“程序设计＝算法＋数据结构”这一著名公式，代表了面向过程的结构化程序设计的基本观点和方法。在有限元程序设计等数值计算领域中，人们也广泛使用这种格式。以结构程序设计为代表的面向过程的程序设计方法的成功不是偶然的，而是软件工程学历史发展的结果。

40、50年代电子数字计算机问世之初，人们是用机器语言和汇编语言编写程序的。这种程序要求为完成一个任务需对计算机进行的内部操作作详尽的描述。这一时期的特点是：一方面低级语言对计算机操作的具体指令使得掌握这类语言非常困难。用这些语言进行程序设计是一项个人的浩繁工作。开发周期长，劳动效率低，程序调试和维护困难。另一方面低级语言对具体计算机的依赖使程序几乎没有通用性和可移植性。在这一时期，计算机程序设计只能是极少数专家的高深的工作。除了应具有特定专业领域的知识外，这样的专家还要有低级语言和计算机硬件方面的专门知识。这样，由于在计算机和它的潜在应用领域之间不存在“用户界面”，即按照人们常规思维方式进行程序设计的工具，无法从计算机科学中独立出程序设计这一为非计算机专业人员所有的领域，大多数人对计算机程序设计只能望而却步。

这种对计算机普及和应用的严重障碍促使人们研究能体现人的逻辑推理、与计算机内部操作无关的程序设计语言，即高级语言。50年代后期到60年代中期Fortran等高级语言的出现，标志着计算机应用技术从极少数专家手里解放出来而面向工程、科学和社会生活各个领域的一次革命性转变。从这个时期开始，计算机软、硬件的开发和应用的关系就基本上固定了下来。即

硬件开发→操作和编译系统开发→应用软件开发→用户

正是由于高级语言彻底摆脱了对具体计算机及其硬件环境的依赖，具有自然语言和数学语言的特点，才促使了广大应用软件开发人员在各自的专业领域内进行应用软件开发，从而使计算机应用迅速推广到科学研究、工程技术、文字和图象处理、金融财会、档案管理、人工智能和专家系统等科学技术和社会生活的各个方面。即使只有中等文化水平的人，只要经过一两个月的学习，也能够用高级语言在一定的范围内进行程序设计。从此，计算机程序设计再也不是令人生畏的少数专业人员的繁重个人劳动，计算机的应用得到了空前的普及。

然而，这一时期的软件开发并无系统的方法可循。软件开发只不过是编写程序（即代码）的代名词而已。与当时的计算机硬件、软件系统相适应，人们所关注的，往往是节省几条指令或几个存储空间这样的怎样用个人的技巧来提高程序效率的具体问题。

60年代后期，数据库、操作系统、数值计算软件包等大型软件系统开始出现。需要人们开发的软件系统往往含有数万到上千万条指令。软件系统的规模和复杂性的剧增，引发了人们不曾预见到的“软件危机”。一方面，可达成百上千人年工作量的软件系统，必须作为产品组织一批人来协同开发，需要大量的投资。另一方面，在只讲程序设计个人技巧和表达能力，缺乏统一的软件开发方法制约的传统观念约束下开发出来的软件产品，其质量、稳定性、可

重用性、可维护性等方面都不可避免会存在重大隐患。诸如一个程序员的偶然错误可使整个系统崩溃等问题使大型软件系统的开发面临几乎不可克服的障碍，以至有人用“在泥潭中作垂死挣扎的巨兽”来形容这一时期的软件开发工作。

这一严重的“软件危机”，使人们认识到软件开发必须工程化、规范化、系统化。60年代以来逐渐形成和完善的软件工程学，就是在这一背景下诞生的。它要回答的，主要是下列问题：

1）软件开发过程阶段性的划分。必须把大型软件系统的开发划分为若干个主要的阶段，并对每一阶段的任务和工作步骤制订标准规程，以指导和约束软件开发人员的设计方案和代码编写。

2）软件产品质量的检查制度和手段。为了保证软件开发每一阶段的质量，防止错误的阶段性传播，必须有有效的检查验收手段和制度。

3）完整的软件开发技术档案的建立。软件开发人员良好的素质和习惯，如在源代码中提供尽可能多的说明和遵守某些使代码具有较高易读性的约定固然是建立软件开发技术档案的组成部分，但更重要的，是建立软件开发每一阶段的文档。文档主要有两类：提供给专业人员和验收人员的技术指标和操作说明，以及提供给用户的使用说明。技术档案的建立，有利于软件的验收、维护和推广。

软件工程学这三项任务的基础，是软件开发过程阶段性的划分与实施。

结构程序设计，就是逐渐发展和完善起来的为完成这一根本任务的重要方法。它有三个主要特点：模块化、自顶向下和逐步求精。

模块化是指将一个大型的程序分解为相对独立的程序段，即模块。每一模块完成某种特定的操作，可单独设计和调试。模块之间通过界面进行信息交换。模块化的主要性质可以归纳为两条。第一是局部性，对程序某一局部（通常是模块内部）的改变不影响其余部分。从而可以有效地防止错误的扩张，提高软件系统的稳定性和可维护性。第二是抽象性，人们可以使用一个模块而无须知道它如何工作。这就使得大型软件系统的按层次分工开发和组装的工业化生产方式成为可能。

自顶向下和逐步求精实际上是实现模块分解的具体方法：结构程序设计的步骤，就是从程序的抽象分析开始，到具体的实施细节为止。

历史上，曾经证明，只要用顺序、选择和循环这三种基本控制结构，就可以设计出任何单入口、单出口的复杂程序。实际上，无论是模块内部设计，或者是模块控制信息，都是用这三种结构元素的交错嵌套来实现的。

经过人们二十多年不懈的努力，结构程序设计方法已得到充分的发展和应用，为实现软件工程学提出的软件开发工程化、规范化、系统化的任务，提供了一个实用的方案。

在我国，国家标准局在1988年颁布的两个关于软件开发规范的国家标准GB8566—88《计算机软件开发规范》[1]和GB8567—88《计算机软件产品开发文件编制指南》[1]，就是把结构程序设计模块化、自顶向下和逐步求精的方法标准化并强制实施的产物。

正是在这种背景下，“程序设计＝算法＋数据结构”这一公式表明了结构程序设计方法在软件工程学中的巨大成功和重要地位。

然而，结构程序设计方法也有其局限性。计算机技术是描述客观世界的多样性和解决人们关心的各类问题的有力工具，但客观世界并不是算法和数据结构所能包容的。结构程序设计方法并没有完全解决超越程序复杂性和自然地表示客观世界的问题。

近年来发展起来的面向对象系统分析技术和程序设计方法代表了崭新的计算机程序设计的思维方式和具体方法。它是结构程序设计、信息隐蔽、数据封装、知识表示[2]、并行处理等各种基本概念和方法的综合，是最有潜力解决克服程序复杂性障碍和自然表示客观世界这两个问题的工具。面向对象技术的基础是对象，它表示一个封装数据和操作的实体。具有相同性质的对象抽象为类。类由继承关系组成面向对象的系统。与结构程序设计不同的根本之处在于，是对象而不是过程或函数作为问题的中心环节。结构程序设计数据结构的主要弱点，是过程和函数的运行严格受控于基本数据结构。基本数据结构的微小改变影响整个软件系统，所涉及的所有过程和函数必须修改甚至重写。当关键数据结构必须改变时，导致整个系统的崩溃。在面向对象技术中，软件系统的构造不依赖于对象内部的数据结构，而仅取决于对对象进行操作的方法。

面向对象的系统分析技术和程序设计方法一经出现，就显示了强大的生命力。经过数年的发展，面向对象的程序设计语言、面向对象的操作系统、面向对象的数据库、面向对象的开发环境和工具，已取得了系列成果。有些产品已投入商业化使用。虽然预言面向对象技术将会替代面向过程技术（以结构程序设计为代表）还为时太早，而且这一技术本身还在不断发展和完善中，但它的强大生命力和在软件工程学与程序设计方法学中划时代的作用，已经明显地表现出来。

作为求解复杂工程问题的有限元方法，是与结构程序设计方法同步发展的。因而有限元法有幸使用结构程序设计方法作为有力的开发工具。但是复杂的控制结构和数据结构阻碍了有限元方法的发展和程序维护，使得开发的成本越来越高。

规模的扩大，代码的重用性、程序的可移植性、可扩充性和可维护性，已逐步成为有限元软件开发的主要任务之一，并占有绝大部分工作时间。用传统的以算法为核心、过程和数据分离的面向过程的程序设计方法和语言来完成这些任务，已成为有限元方法发展的沉重负担。另一方面，传统的有限元程序结构不适于知识表达。要加入知识获取与处理机制，实现有限元方法智能化是困难的。

面向对象技术为解决困扰有限元发展的问题提供了方案。正如在非数值领域取得的巨大成功一样，面向对象技术在数值计算领域也具有巨大的潜力。

当然，一个问题是：面对有成百万条指令，已投入运行的有限元商业软件包，面对大学和研究所自成体系的有限元程序，面向对象技术能够做些什么呢？

回答是肯定的。面向对象方法使得人们能对现有的用结构程序设计方法开发的软件作系统分析，并在不花费太大成本的条件下将其改造为面向对象系统，从而最大限度地利用现有的资源。

本书将介绍面向对象系统分析和程序设计方法的基础，以及 Microsoft C++7.0 语言的要点。通过对一个 Fortran 有限元软件包的改造，逐步介绍面向对象技术在有限元分析中的应用。书中第 3～8 章介绍的面向对象 C++有限元软件包由西南交通大学出版社出版发行。读者还将看到如何在面向对象的有限元系统中增加新的单元类和新的功能。

面向对象程序设计是一个正在发展中的方法。其在数值计算领域的应用，更是一个新的领域。限于作者的水平，本书难免有错误之处，祈望读者不吝指正。

孔祥安

1994 年 12 月于成都

目　录

第一章　面向对象程序设计

§1.1　面向对象程序设计的要点

在传统的面向过程或模块化的程序设计方法看来，一个程序就是描述一系列要计算机施行的操作，即某种算法。按面向对象程序设计的观点，一个程序所描述的，是“对象”相互作用的一个系统。

面向对象程序设计方法建立在三个基本概念上。第一是抽象。抽象可大大地简化大型程序的编写。第二是数据和信息的隐蔽性。隐蔽性使得程序易于修改和维护。第三，也是最重要的，是类的继承性，一种对程序功能和信息传递的强有力的分类工具。类的概念和继承性使得程序的可扩充性从根本上得到改善。

面向对象程序设计的这三个基本概念回答了我们在前言中提到的结构程序设计难于解决的几个问题。从程序设计方法学的观点看来，原则上用任何高级语言都可以实现抽象性、隐蔽性和某种程度的继承性，但只有面向对象的程序设计语言，例如C++，才从语法上针对这些原则作出了明确的规定。

§1.1.1　抽　象

“抽象”是指集中在问题的基本特性上而忽略某些细节。如果一种程序设计语言支持高度的抽象，我们就说这种语言是高水平的。例如一个任务既可以用汇编语言，也可以用C语言来完成。汇编语言程序对计算机要执行的每一步操作都作详细的描述。然而人们通常并不关心在这一层次上发生的事情。C或其它高级语言程序却是对计算机操作的高度抽象。这种抽象使得写出来的程序既简洁又易于理解。

虽然传统的高级语言也支持抽象的概念，但面对对象程序设计语言进一步提供了更高度抽象的功能。以下我们从对过程的抽象和对数据的抽象这两方面来说明这一点；并由此引出为面向对象程序设计所特有的“类”的概念。

·过程抽象

过程抽象是抽象的最一般形式。其目的在于使程序员集中于要点而不被细节所干扰。过程抽象的最好例证莫过于大多数高级语言都支持的自定义函数、过程或子程序。定义这些函数、过程或子程序，实际上就是对程序的过程抽象。这一方面可以举出许多例子。在大型数值计算程序中，过程抽象的重要性也是不言自明的。例如，模块化的有限元程序的基础就是过程（procedure）或子程序（subroutine）。过程抽象不仅节约了输入程序的时间，更重要的，还在于使程序更具易读性和共同开发的可能性。实际上，过程抽象使得从高层次上来进行设计成为可能。可能分散注意力和打断程序设计逻辑思维的细节被隐藏起来了。每一个过程或子程序的功能如何实现并不重要，重要的是它在程序框架中的作用及与其它过程或子程序的关系。

·数据抽象

数据抽象的目的，是为了使我们避开计算机构造数据类型的具体方式。数据抽象的概念是比过程抽象更为基本的。所有程序员实际上都享受到数据抽象的巨大好处。例如整型和浮点数就是一种数据抽象，因为计算机实际上是对二进制或十六进制数进行操作的。设想一下没有这一最基本的数据抽象我们该如何工作？幸运的是，整型或浮点数的数据抽象使得一个C或Fortran程序员无须操心他的代数运算是如何用二进制数完成的。这也说明，数据抽象总是包含某种程度的过程抽象的：当我们对某种数据类型的变量进行操作时，我们并不知道（也不需要知道）这种数据的格式。即对这些数据类型的操作细节被避开了。

大多数程序设计语言支持定义新的数据抽象类型的能力是很有限的，Fortran尤其如此。C的struct和typedef支持用户定义的数据类型。但很多情况下，C的结构只是作为变量的堆积而使用。例如：

```
typedef struct
{
 int     ElementNumber;
 int     ElementNodeNumber;
 int     ElementConnectivity;
 float *pElementNodeCoordinate;
 // …
 }
S_ElementInfo;
```

这种用户定义的类型的方便之处在于可将数种具有某种物理、几何或其它方面联系的信息作为一个单元进行统一操作。但是这并未提供新的概念，因为我们不可能只考虑这一结构而不具体涉及它所包含的每一个分量的信息。

在C的STDIO.H中有更好的数据抽象的例子：

```
typedef struct _iobuf
{
 char __far *_ptr;
 int  _cnt;
 char __far *_base;
 char _flag;
 char _file;
 } FILE;
```

这样的FILE（文件）结构从概念上就比它所包含的分量前进了许多。可以只考虑FILE而不必过问其表达方式。只要用一个FILE指针，就可以将对FILE的操作交与各个库函数去施行。

虽然数据抽象和过程抽象是紧密相联的，C语言还是把它们处理为两类不同的功能。因此可以只定义一个结构而不定义使用这一结构的方法。

·类

类（class）的概念是面向对象语言所特有的。类将过程抽象和数据抽象结合在一起。定

义一个类时，也就同时对一个高层次的实体的各方面进行了描述。当引用这个类的一个实例（或称对象，与基本数据类型的变量相对应）时，可以避开在类中所包含的基本类型和所定义的对它们操作的过程。

考虑有限元中可能定义的一个简单的类：平面三节点线性单元。从几何的角度来看，人们也许要把这种单元视为三个点，并作为三对数（坐标）而存储。然而有限元中三节点单元所包含的，却远比只是三个顶点为多。一个单元有周长，有面积和具体的形状。在划分或生成单元时，可能要把一个单元移动、旋转或反射。对两个单元，我们可能要知道它们的交或并。或判断它们是否相等。从物理角度看，可能具有的性质或操作就会更多。利用类的概念，所有这些性质和操作在高层次的实体都是完全有意义的，人们无须在低层次上考虑如何构造这些单元就能办到。

面向对象语言支持的这种对数据抽象和过程抽象的组合，在程序和计算机之间建立了一个十分有意义的新的界面。自定义的高层实体——类——对基本数据类型、结构、联合(union)等所具有的优势就象浮点数对二进制字节，printf 语句对 MOV 指令所具有的优势那样。大型而复杂的应用程序的编写由此可得到极大的简化。

类还可以表征一些通常不被视为数据类型的实体。例如，可以定义一个表征二叉树的类。它的每一个对象并不只简单的是树的一个节点，而本身也是一个树。这样，建立一个多重二叉树就与建立一个单个二叉树一样简单。人们对于一个二叉树感兴趣的性质是：迅速查找、增加、删除某项，及按某种顺序列举所有项的功能。只要能完成一套相同的运算，用何种数据结构是关系不大的。可以用节点和指针构成的树，也可以用数组构成的树，或别的什么数据结构。

这样的一个类可命名为 BinaryTree，这一名字指明了一种具体的植入方式。本书中我们把设计程序的源代码称为植入。而根据对类的可以进行的操作，又可命名为 SortedList 或诸如此类的名字。

以类这样的具有自定义的一套运算的抽象实体为基础，而不是以基本类型构成的数据结构为基础进行程序设计，就能更加独立于植入细节。由此引入面向对象程序设计的另一个优势：数据的封装和隐蔽性。

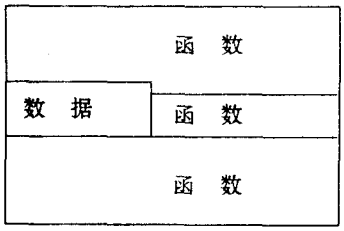
§1.1.2 隐蔽性

数据的封装或隐蔽性是指隐藏一个类的内部工作的细节从而支持或加强抽象性。要实现隐蔽性，就要在类的界面和它的植入之间作一个明确的划分。界面具有公共可见性，即对程序的其它部分也是透明的。类的植入则只具有私有可见性，即只在类的内部才是透明的。类的界面描述这个类做什么，而类的植入描述的是如何去做。为支持抽象性，界面中只对外暴露尽可能少的细节。使用类的对象的人，只须通过界面了解这个类能施行哪些操作。

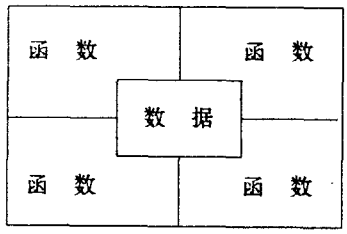
有些文献把隐蔽性解释为将函数和数据组合在一起的作法。这往往会引起误解。因为可以将函数和数据组合在同一个类里并且都作为公有部分。但很明显，这并不能实现隐蔽性。形象地说，真正的具有隐蔽性质的类是将数据藏于其函数之中。或者说用函数把数据封装起来，只能通过调用函数来对数据进行操作。如图 1.1 所示。

隐蔽性并不是面向对象程序设计所独有的。在结构程序设计中，数据隐蔽是用于模块而不是类。模块化就是将一个大程序划分为模块。每一个模块都有明确定义的函数界面来与其

它模块交换数据或相互调用。模块设计的好坏，主要是看它在什么程度上作到了对其它模块的数据结构“一无所知”，以及是否只通过界面来与其它模块发生联系。Fortran 的 COMMON 和 EQUIVALENCE 语句就明显地违反了这一原则。要避免模块之间在数据结构上互相牵制，就必须尽量限制使用全程变量或全程数据结构。



数据作为公有部分的类



数据作为私有部分的类

图 1.1 用函数封装数据

例如，要设计一个程序来对一个信息表进行操作。用 C，所有与信息表有关的函数可以定义在一个程序块 LIST.C 里，它们的原型在头文件 LIST.H 中说明：

LIST.H

```
#include "struct.h"      //  get definition of S_STR data type
#include "macro.h"       //  get definition of P_1(), P_2() macros

ERC     ListAddHead      P_2( S_LISTITEM *pListItem, void *p );
void    ListGetIndex     P_2( void *pThis, short int *pIndex );
/*   ... other functions  */
```

程序中任何函数在用到信息表时，都可以调用在 LIST.H 中说明的函数。至于在 LIST.C 中植入信息表的方式，可以选用数组，但程序的其它模块对此并无所知。而如果这个数组说明为 static，则只能在 LIST.C 中访问它。这就是说，只有界面是可见的，而植入则是完全隐蔽的。

数据隐蔽有若干优越性。其一是抽象。这在前面已提到过。抽象性使我们可以使用一个模块而不必考虑它是如何工作的。另一个是“局部性”，即改变程序的一处并不影响其它地方。局部性不强的程序是十分脆弱的。对这样的程序一处作了修改会引起其它程序段失效，因为它们都是互相依赖的。而具有良好局部性的程序是稳定的，是易于维护的；改变一段程序的影响被局限于一小部分。例如，如果使 LIST.C 中的链表变为数组或别的数据结构，并不会影响到使用 LIST.C 中函数的模块。

另一方面，用模块来隐蔽数据也有局限性。上面例子中的 LIST 模块并不能允许程序中使用多个信息表，也不能允许定义一个局部于某一个函数的信息表。可以用结构和指针来实现这些性质。例如，把指针作为信息表的句柄，然后把信息表指针作为函数的参数：

LIST.H

```
#include "struct.h"
#include "macro.h"
```

```
/*   define S_LIST with a typedef */
ERC   ListInitialize     P_1( S_LIST   ** ppListHandle );
void  ListInitDefault    P_1( S_LIST    * pListHandle );
void  ListRelease        P_1( S_LIST   ** ppListHandle );
ERC   ListAddHead        P_3( S_LIST    * pListHandle,
                              S_LISTITEM * pListItem, void * p );
void  ListGetIndex       P_3( S_LIST    * pListHandle,
                              void * pThis, short int * pIndex );
/* ... other functions */
```

这一技术比起前一个例子中 LIST.H 中函数说明来，要有用得多。由此我们可以同时使用多个信息表。然而，这种 LIST 数据类型并不能像基本类型那样方便地使用。例如，在退出函数时，局部信息表并不自动地删除。与动态变量一样，这些信息表需要用专门的程序段来小心地加以管理。

现在，考虑信息表的 C++植入（关于类定义的语法，请参见本书第二章）：

```
// LIST.H
#include "struct.h"
#include "macro.h"

class List
{
public:
 List();
 ERC   ListAddHead      P_2( S_LISTITEM   * pListItem, void *p );
 void  ListGetIndex     P_2( void * pThis, short int * pIndex );
~List();
private:
 //...
 };
 //...

// PROG.CPP
//#include "list.h"

void fun()
{
 List myList, yourList;
 //...
}
```

与 C 程序的 pListHandle 句柄进行的操作相比较，这一 C++的类有两个优势：

第一，使用方便。我们可以定义 List 的一些实例，就像定义整型或浮点数的变量那样方便。所有实例的作用域规则都是相同的。

第二，类的定义加强了隐蔽性。这是更为重要的一点。很显然，在使用指针时不对 pListHandle 句柄中的数据进行操作只是程序员之间的约定。然而，许多程序员为图方便却会选择越过函数的界面而对句柄直接操作。这样的程序员对他的数据结构的规定一般会涉及程序的许多地方。一旦一个信息表的植入有所改变，显而易见必须在程序中找出不再适用某些规定的所有地方，并作相应的改动。这当然是非常繁琐和易出错的。而这样的错误编译时往往不会发现，而只有到了运行程序时才表现出来。例如空指针的引用。即使对程序很小的改动也可能引发这样的问题。往往有这种情况：为纠正错误而修改程序，但却带来新的错误，因为别的函数也依赖于同一种植入的方法。

相反，把信息表定义为一个类，就可以用面向对象语言的访问规则来隐蔽植入。我们可以不必担心那些只图省事的程序员。任何试图访问类的私有数据（private）的程序都通不过编译。程序的局部性从而得到更好的保证。

程序员违反约定对某种数据结构直接操作的最一般原因，在于用界面中的函数实现起来要麻烦一些。一个设计良好、准确反映类的主要特性的界面固然可以减少一些这种麻烦，但任何界面都难以保证所有的操作都十分方便。然而禁止访问类的内部数据结构的语法规定是十分必要的，尽管有时这是以一段效率不高的程序为代价的。与隐蔽性带来的对程序维护的加强相比，这一在“方便性”方面的微小损失实在算不了什么。面向对象的语言从语法上消除了在对程序的局部作改动时可能会触及许多其它模块的缺点，这就从根本上减少了开发新的程序系统和更新现存系统的代价。

即使类的界面以后可能改动，也应该使用隐蔽的类而避免可访问的数据结构。多数情况下，这些改变可由只在现有界面上附加新的内容而实现。这就保证了程序更新时的相容性。任何使用老界面的程序仍可正确地照样工作。当然，程序需重新编译，但这消耗的是计算机的时间，而不是程序员的时间。

需要提醒的是，隐蔽性并没有杜绝从外部对类的私有数据访问的可能性。程序员可以用 & 和 * 算符来做到这一点。隐蔽性防止的，只是那些偶然地介入类的内部表示的企图。

§1.1.3 类的继承性

继承性的概念是所有过程性程序设计所没有，而为面向对象程序设计所独有的。C++语言允许从一个类导出另一个类，从而引入基类和子类的概念。若干类可能具有某种相似性，或者共属于一个更广泛范畴的子范畴。在 C++里，它们都可以从一个基类导出；或者可以在它们的基础上建立一个包含共同性质的基类。相反，在 C 或 Fortran 中，所有的数据类型都是完全独立的。

在几个类的基础上建立一个共同的基类也是一种抽象的方式。基类是这些类的更高层次的综合，它概括子类共有的东西。因而基类使人集中考虑这些共性而暂时忽略各个子类的个性。这种抽象的技术使人们可以根据少量的关键性质而不是大量罗列的琐碎性质来考察程序设计中的实体。

如果说基类是一组类的概括的话，那么子类就是基类的特殊化，或特例。子类表征一个已经定义的类的子类型，它描述的是这一个类的附加性质。

子类有时也称导出类。基类和导出类构成类的层次。在这一层次中，类的继承性有两个优越性：导出类既可共享基类的代码，也可共享基类的界面。虽然为代码重用而设计的继承性与为界面重用而设计的继承性具有各自的特点，但它们并不是截然分开的。

·代码继承性

如果一个新的类要使用一个已有的类的功能，那么可以简单地从这个类导出。在这种情况下，继承性允许代码的重用。

在同时需要若干个具有一些共同特性的类时，继承性可以避免大量的代码编写。只须将这些共性一次编码并植入一个基类，而不必在每一个类中重复这些代码。

例如，要设计一个读入有限单元信息的程序，让用户输入单元信息的域。这些域可能包含节点信息，单元信息，边界条件信息等等。每一个域只接受适当的数据类型。可以将每一个这样的域作为一个单独的类，分别取名为 NodeField、ElementField、BoundaryConditionField 等。每一个类有各自的检查合法输入的判据。然而，所有这些类都有一些共有的功能。每个类都要提示用户需输入什么，显示和确认输入结果的功能也是相同的。于是所有这些类都有相同的 SetPrompt()、Display Prompt()等函数。

这样，通过定义一个称为 Field 的类来植入这些公共的功能可以节省编程工作量。NodeField、ElementField 和 BoundaryConditionField 类都可直接从 Field 导出。这里体现的类的继承性，同时减少了查错和增加新功能的工作量，因为只需在一处对程序进行修改。

为代码共享而进行继承性设计的原则是：尽可能将代码置于基类，即靠近类的继承层次的顶部。这样，基类的代码就可以为许多导出类所重用。因此可以说，导出类是基类的特殊版本或扩充版本。注意在类的继承关系中，基类和导出类可能不止一个层次。处于继承关系中间的类既是基类，又是导出类。

·界面继承性

在面向对象程序设计中，界面继承性是指导出类只继承基类函数（或称为成员函数）的名字；这些函数的代码是在导出类给出的。由此，导出类与基类具有相同的界面，用相同的函数名各执行不同的功能。

界面继承性的设计思想使不同的类使用相同的界面，从而增强了这些类的属性在高层的相似性。界面继承性的主要优点是由多态性表现出来的。下面我们就来看一个例子。

有限元方法中要计算数值积分点的坐标和权函数值。我们用一个函数 GAUSS()来实现。常用单元按数值积分函数性质可分为四边形（含一维的二节点、三节点单元，二维的四节点、八节点单元，三维的八节点、廿节点单元）单元、三角形单元、棱柱单元和四面体单元。我们定义一个称为 Geometry 的基类，Rectangle、Triangle、Prismatic 和 Tetrahedral 作为导出类。导出类的实例作为广义的 Geometry 对象操作。具体地说，类 Geometry 有一个成员函数 GAUSS()，但这一函数除了名字外并无任何内容。Rectangle 类继承了这一成员函数并给出对广义的四边形单元的数值积分点坐标和权函数。Triangle、Prismatic 和 Tetrahedral 类里的同名函数完成相同的事情，只不过对不同类型的单元而已。虽然这些类的对象显示不同的属性，但它们都共享相同的界面。于是在高层上，它们都可视为 Geometry 类的对象。这里的继承关系可图示如下（见图 1.2）。

在第五章 § 5.3 中，我们还要详细讨论类 Geometry 及其继承关系。Geometry 类是一个抽象的基类，其子类只继承它的界面。前面提到的读入有限单元信息的程序可以设计为既继承

界面又继承代码。

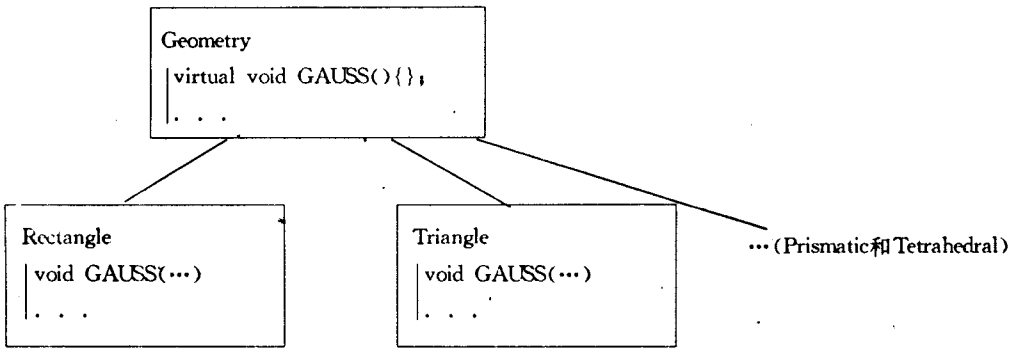


图 1.2　类 Geometry 的继承关系

为界面共享而进行继承性设计的原则是：尽可能将代码置于导出类，即靠近类的继承关系的底部。导出类给出基类中定义的抽象模型的具体形式。

总起来说，面向对象程序设计中类的引入，提供了对抽象性、隐蔽性和继承性的全面支持。类是一个定义抽象数据类型及对这些类型的操作的工具。类可以被封装。利用类可以加强程序的局部性，改善程序的结构。可以把类组织为继承的层次，以此突出类之间的逻辑关系，从而最大限度地减少在过程语言程序设计中不可避免的重复或多余的代码。

C++语言提供了面向对象程序设计的工具。当然，用C++语言也能进行非面向对象的程序设计。下面我们简要介绍如何设计一个面向对象的程序系统。

§1.2　面向对象程序系统的设计

我们知道，结构程序设计是自顶向下的。结构程序设计中的第一步是要确定程序的功能。因此，要程序“做什么”是要明确的首要问题。一般的做法，是先用高度抽象的伪码或流程图来描述全局性的步骤。其中实现有些步骤的方法可能马上解决，大多数则尚须进一步分解。这样层次性地向下发展直到所有步骤得以实现。这种技术称为过程分解。形象地说，就是把一个大的问题描述为一个工序，再逐步细分为子工序。

面向对象设计方法与这种技术很不相同。在面向对象设计中，不必根据任务或工序来分析问题。也不必根据数据来描述问题，即回答“程序建立在什么数据结构上”来开始设计。相反，面向对象设计方法按对象的相互作用来提出和分析问题。第一个问题就是：需要什么对象，或者程序中需要哪些实体。

不但面向对象设计的起点与过程分解不同，设计展开的方法也不一样。过程分解是一种自顶向下的方法，即始于对程序的抽象描述，止于实现的具体细节。面向对象设计一般不采用这一自顶向下的技术。程序员不必首先定义一些大类，然后将它们细分为小类。同时，也不能说面向对象技术就是自下向顶的，即从小类开始建立大类。面向对象设计是在设计的所有步骤中同时在各个层次进行的。

面向对象程序设计要求程序员完成下列工作：

1）识别可能的类，或潜在的类

2）赋予类属性和行为

3）找出类之间的关系