第 1 章　数据结构概论

数据结构有用吗？当然有用，现在只要使用计算机，都会涉及数据结构，特别是从事计算机系统开发的专业人员，必须切实掌握数据结构的知识，它不仅是进入计算机行业的敲门砖，而且是开展计算机业务的技术基础。

数据结构难学吗？其实不难，只要仔细读书，认真思考，就能切实掌握各种结构的本质和它们之间的关系。有人说数据结构太难学，实际上是指动手编写程序难。这个问题是无法回避的，没有这方面的训练，将来就无法胜任系统开发的工作。动手能力的提高，需要日常无数次练习和总结。与学习任何一门课程一样，要踏踏实实地做题和实验，不断提高自己的编程能力和算法设计能力。

1.1　数据结构的概念

1.1.1　数据结构举例

众所周知，无论是使用计算机进行工程和科学计算，还是使用计算机做工业控制或信息管理，都属于数据处理的范畴。那么，如何在计算机中组织、存储、传递数据，就成为一个必须解决的问题。下面看几个例子。

【例 1】 在一个学生选课系统中，有两个数据实体（现实世界中的事物），即“学生”和“课程”，形成了两个数据表格。其中的“学生”实体包括了许多学生记录，它们按照每个学生学号递增的次序，顺序存放在“学生”表格中；而“课程”实体包括了各个课程记录，每个课程也有个课程编号，在“课程”表格中各个课程按照其课程编号递增的次序依次排列，如图 1.1 所示。

“学生”表格

	学 号	姓 名	性别	籍 贯	出生年月
1	98131	刘激扬	男	北京	1979.12
2	98164	衣春生	男	青岛	1979.07
3	98165	卢声凯	男	天津	1981.02
4	98182	袁秋慧	女	广州	1980.10
5	98203	林德康	男	上海	1980.05
6	98224	洪　伟	男	太原	1981.01
7	98236	熊南燕	女	苏州	1980.03
8	98297	宫　力	男	北京	1981.01
9	98310	蔡晓莉	女	昆明	1981.02
10	98318	陈　健	男	杭州	1979.12

“课程”表格

	课程编号	课 程 名	学时
1	024002	程序设计基础	64
2	024010	汇编语言	48
3	024016	计算机原理	64
4	024020	数据结构	64
5	024021	微机技术	64
6	024024	操作系统	48
7	024026	数据库原理	48

图 1.1　学生选课系统中的两个数据实体

在“学生”表格中，各个学生记录顺序排列，形成一个学生记录的线性序列，每个记录在序列中的位置有先后次序，它们之间形成一种线性关系。“课程”表格中的情况完全相同。

【例 2】 在学生选课系统中，一个学生可以选修多门课程，一门课程可以被多个学生选修，这在“学生”和“课程”实体之间形成多对多的关系。为了便于处理，引入一个新的实体“选课”，它的每一个记录是一张选课单，包含如下信息：(学号，课程编号，成绩，时间)。

此时，在“学生”实体、“课程”实体和“选课”实体间形成如图 1.2 所示的关系。这是一种网状关系。学生对选课、课程对选课都是一对多的关系。

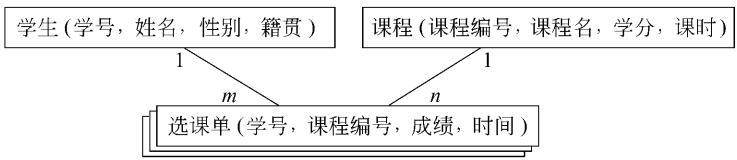


图 1.2 学生选课系统中实体构成的网状关系

【例 3】 一个典型的 UNIX 文件系统的系统结构如图 1.3 所示。这是一个层次结构：在此系统结构图中，顶层结点代表整个系统，用根目录/表示；它的下一层结点代表系统的各个子系统，即根的子目录，如/bin、/lib、/user 等；再下一层结点代表更小的子目录，如/user/yin、/bin/ds，如此类推，直到底层，即为文件，如/user/yin/queue.cpp。

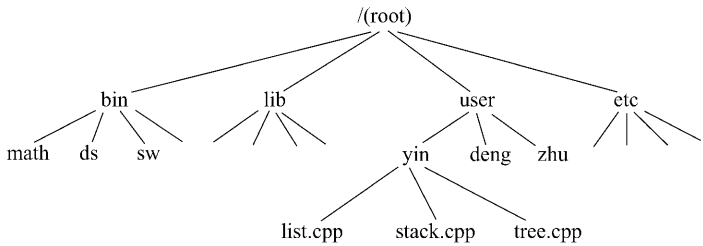


图 1.3 UNIX 文件系统的系统结构图

综上所述，在应用程序中涉及各种各样的数据，为了存储它们，组织它们，需要讨论它们的归类及它们之间的关系，从而建立相应的数据结构，并依此实现要求的软件功能。本课程的目的就是为系统开发者提供解决此问题的基本知识，使得读者在学习完本课程之后，能够在系统开发时运用学过的知识，在设计系统体系结构的同时设计出有效的数据结构，正确、高效地实现整个系统。

1.1.2 数据与数据结构

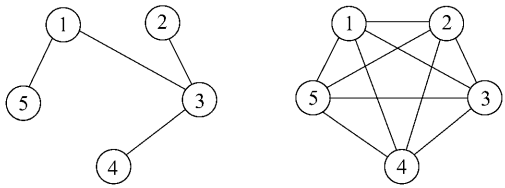
人们在日常生活中会遇到各种信息，如用语言交流的思想、银行与商店的商业交易、在战争中用于传递命令的旗语等。这些信息必须转换成数据才能在计算机中进行处理。因此，需要给数据下一个定义：

数据(data)是信息的载体，是描述客观事物的数、字符，以及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。数据大致可分为两类，一类是数值性数据，包括整数、浮点数、复数、双精度数等，主要用于工程和科学计算，以及商业事务处理；另一类是非数值数据，主要包括字符和字符串，以及文字、图形、图像、语音等数据。

数据的基本单位是数据元素(data element)。一个数据元素可由若干个数据项(data item)组成,它是一个数据整体中相对独立的单位。例如,对于一个学籍管理文件来说,每个学生记录就是它的数据元素;对于一个字符串来说,每个字符就是它的数据元素;对于一个数组来说,每一个数组成分就是它的数据元素。

在数据元素中的数据项可以分为两种,一种叫做初等项,如学生的性别、籍贯等,这些数据项是在数据处理时不能再分割的最小单位;另一种叫做组合项,如学生的成绩,它可以再划分为物理、化学等更小的项。通常,在解决实际应用问题时把每个学生记录当作一个基本单位进行访问和处理。

在数据处理中所涉及的数据元素之间都不会是孤立的,在它们之间都存在着这样或那样的关系,这种数据元素之间的关系称为结构。例如,招生考试时把所有考生按考试成绩从高到低排队,所有考生记录都将处在一种有序的序列中;又例如,在 n 个网站之间建立通信网络,要求以最小的代价将 n 个网站连通,如图 1.4(a)所示,这样,在所有网站之间形成一种树形关系;反之,要求网络中任一网站出现故障,整个网络仍然保持畅通,这样,在所有网站之间形成一种网状关系,如图 1.4(b)所示。



(a) 网站之间的树状关系　　(b) 网站之间的网状关系

图 1.4　n 个网站之间的连通关系

由此可以引出数据结构的定义:**数据结构由某一数据元素的集合和该集合中数据元素之间的关系组成**。记为:

$$\text{Data\_Structure}=\{D,\ R\}$$

其中,D 是某一数据元素的集合,R 是该集合中所有数据元素之间的关系的有限集合。

注意,有关数据结构的讨论主要涉及数据元素之间的关系,不涉及数据元素本身的内容。关于数据元素的内容,在系统开发时考虑。

1.1.3　数据结构的分类

依据数据元素之间关系的不同,数据结构分为两大类:线性结构和非线性结构。

1. 线性结构(linear structure)

线性结构也称为线性表,在这种结构中所有数据元素都按某种次序排列在一个序列中,如图 1.5 所示。对于线性结构类中每一数据元素,除第一个元素外,其他每一个元素都有一个且仅有一个直接前驱,第一个数据元素没有前驱。除最后一个元素外,其他每一个元素都有一个且仅有一个直接后继,最后一个元素没有后继。

图 1.5　线性结构中各数据成员之间的线性关系

根据对线性结构中数据元素存取方法的不同，又可分为直接存取结构、顺序存取结构和字典结构。对于直接存取结构，可以直接存取某一指定项而不须先访问其前驱。像数组、文件都属于这一类。可以按给定下标直接存取数组中某一数组元素；可以按记录号直接检索记录集合或文件中的某一记录。对于顺序存取结构，只能从序列中第一个数据元素起，按序逐个访问直到指定的元素。像一些限制存取位置在表的一端或两端的表（如栈、队列和优先级队列等）就是这种情况。字典与数组有类似之处，但数组是通过整数下标进行索引，而字典是通过关键码（key）进行索引。我们设定数据元素中某一数据项或某一组合数据项为关键码，通过关键码来识别记录。例如，对于学生记录，可设定学生的学号为关键码，用它来识别是哪一位学生的记录。

2. 非线性结构（nonlinear structure）

在非线性结构中各个数据元素不再保持在一个线性序列中，每个数据元素可能与零个或多个其他数据元素发生联系。根据关系的不同，可分为层次结构和群结构。

层次结构（hierarchical structure）是按层次划分的数据元素的集合，指定层次上元素可以有零个或多个处于下一个层次上的直接所属下层元素。树形结构就是典型的层次结构。树中的元素叫做结点。树可以为空，也可以不为空。若树不为空，它有一个叫做根的结点，其他结点都是从它派生出来的。除根以外，每一个结点都有一个处于该结点直接上层的结点。树的结构如图 1.6 所示。

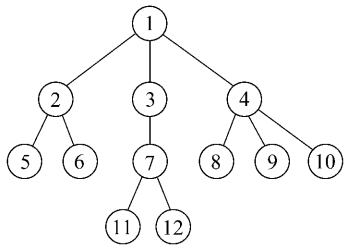
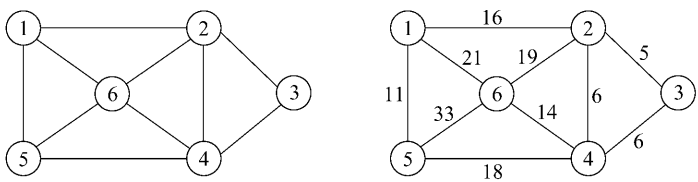


图 1.6　树形结构

注意，有的教科书规定树不能是空树，它至少必须有一个结点，而另外规定 k 叉树（即最多有 k 叉的有序树）可以是空树，并因此推定“k 叉树不是树”。本书则对树与 k 叉树不加区分，在概念上将它们统一起来。

群结构（group structure）中所有元素之间无顺序关系。集合就是一种群结构，在集合中没有重复的元素。另一种群结构就是图结构，如图 1.7(a)所示。它是由图的顶点集合和连接顶点的边集合组成。还有一种图的特殊形式，即网络结构。它给每条边赋予一个权值，这个权值指明了在遍历图时经过此边时的耗费。例如在图 1.7(b)中，顶点代表城市，赋予边的权值表示两个城市之间的距离。



(a) 图结构　　(b) 网络结构

图 1.7　群结构

1.1.4　数据结构课程的内容

数据结构与数学、计算机硬件和软件有十分密切的关系。“数据结构”是介于数学、计算

机硬件和计算机软件之间的一门计算机科学与技术专业的核心课程，是高级程序设计语言、编译原理、操作系统、数据库、人工智能等课程的基础。同时，数据结构技术也广泛应用于信息科学、系统工程、应用数学以及各种工程技术领域。

社会的发展，要求使用计算机解决更复杂的问题，而更复杂的问题需要更大的计算量，从而要求计算机程序的运算速度更快。这就要求计算机专家必须学习和理解隐藏在高效程序设计背后的数据组织和相关算法的效率。“数据结构”课程就是研究系统开发过程中有关设计（包括数据设计、体系结构设计、接口设计和过程设计）的若干基本问题的学科。选择不同的数据结构可能会产生很大的差异：同样一个程序，选择某一种数据结构可能在几秒钟内运行完成，而选择另一种数据结构则可能需要几天时间才能运行完毕。因此，在选择数据结构解决特定问题时，需要预先分析问题来确定必须达到的性能目标，才有可能选出合适的数据结构。如果忽视了这一分析过程，直接选用自己习惯使用的、但与问题不相称的数据结构，就有可能设计成低效的程序；相反，当使用简单的设计就能达到性能目标时，选用复杂的数据表示来改进这个程序也是没有道理的。

当为解决某一问题而选择数据结构时，应当执行以下几个步骤：

（1）分析问题，确定算法遇到的资源限制（内外存空间限制和执行时间限制）。

（2）确定必须支持的基本运算，度量每个运算所受到的资源限制。基本运算包括向数据结构插入一个新数据项，从数据结构中删除一个数据项和搜索指定的数据项。

（3）选择最接近这些资源开销的数据结构。

根据这 3 个步骤选择数据结构，实际上贯彻了一种以数据为中心的设计观点。

为了构造出好的数据结构及其实现，必须考虑数据结构及其实现的评价。因此，数据结构的内容包括 3 个层次的 5 个“要素”，如表 1.1 所示。

表 1.1 “数据结构”课程内容体系

方面 / 层次	数据表示	数据处理
抽象	逻辑结构	基本运算
实现	存储结构	算法
评价	不同数据结构的比较及算法分析	

数据结构的核心技术是分解与抽象。通过分解可以划分出数据的层次（数据—数据元素—数据项）；再通过抽象，舍弃数据元素的具体内容，就得到数据的逻辑结构。类似地，通过分解将处理要求划分成各种功能，再通过抽象舍弃实现细节，就得到运算的定义。上述两个方面的结合将问题变换为数据结构。这是一个从具体（即具体问题）到抽象（即数据结构）的过程。然后，通过增加对实现细节的考虑进一步得到存储结构和实现运算，从而完成设计任务。这是一个从抽象（即数据结构）到具体（即具体实现）的过程。熟练地掌握这两个过程是“数据结构”课程在专业技能培养方面的基本目标。

数据的逻辑结构，也简称数据结构，是指从解决问题的需要出发，为实现必要的功能所建立的数据结构，它属于用户的视图，是面向问题的，如在招生系统中建立的按考分排列的考生记录的有序表格。而数据的物理结构是指数据应该如何在计算机中存放，是数据逻辑结构的物理存储方式，是属于具体实现的视图，是面向计算机的。数据的逻辑结构根据问题

所要实现的功能建立，数据的物理结构根据问题所要求的响应速度、处理时间、修改时间、存储空间和单位时间的处理量等建立，是逻辑数据的存储映像。

通常在课程中讨论数据结构，既要讨论各种在解决问题时可能遇到的典型的逻辑结构（在本书下文中简称为数据结构），还要讨论这些逻辑结构的存储映像（在本书下文中简称为存储结构），此外还要讨论这种数据结构的相关操作（基本运算）及其实现。

数据结构的存储结构可以用以下 4 种基本的存储方法得到：

（1）顺序存储方法（sequential storage） 该方法把逻辑上相邻的元素存放到物理位置上相邻的存储单元中，数据元素之间的逻辑关系由存储单元的邻接位置关系来体现。由此得到的存储表示称为顺序存储结构。通常，顺序存储结构可借助程序语言中的一维数组来描述。

（2）链接存储方法（linked storage） 该方法不要求逻辑上相邻的元素在物理位置上也相邻，元素之间的逻辑关系由附加的指针指示。由此得到的存储表示称为链式存储结构。通常，链式存储结构要借助程序语言中的指针类型来描述。

（3）索引存储方法（indexed storage） 该方法在存储元素信息的同时，还建立附加的索引表。索引表中每一项称为索引项，索引项的一般形式是：（关键码，地址）。关键码是能够唯一标识一个结点（即元素）的那些数据项。若每个结点在索引表中都有一个索引项，则该索引表称为稠密索引（dense index）；若一组相邻的结点在索引表中只有一个索引项，则该索引表称为稀疏索引（sparse index）。稠密索引中索引项中的地址指示结点所在的物理位置，稀疏索引中索引项中的地址指示一组相邻结点的起始存储位置。

（4）散列存储方法（hashing storage） 该方法的处理方式是根据结点的关键码通过一个函数计算直接得到该结点的存储地址。

上述 4 种基本的存储方法既可以单独使用，也可以组合起来对数据结构进行存储映像。同一种逻辑结构采用不同的存储方法，可以得到不同的存储结构（表示）。选择何种存储结构来表示相应的逻辑结构，要视不同的资源要求而定，主要考虑运算的时间和空间要求以及算法的简单性。

1.2 数据结构的抽象形式

1.2.1 数据类型

类型是一组值的集合。例如，布尔（Boolean）类型由 true 和 false 这两个值组成，整数也构成类型，若采用 2 个字节，则整数表示范围在 $-2^{15}\sim2^{15}-1$，即 $-32768\sim32767$ 之间；若采用 4 个字节，则整数表示范围在 $-2^{31}\sim2^{31}-1$，即 $-2147483648\sim2147483647$ 之间。

类型可分为原子类型和结构类型两种。原子类型中的每个数据（即简单数据）都是无法再分割的整体，如一个整数、浮点数、字符、指针、枚举量等都是无法再分割的整体，所以它们所属的类型均为原子类型。结构类型由原子类型按照一定的规则构造而成，如一个银行账户一般包括姓名、地址、账号、余额等原子类型。结构类型还可以包含结构类型，如一个学生的学籍卡片是一个结构类型，它包含的一个数据成分（家庭成员）又是结构类型。所以一种结构类型中的数据（即结构数据）可以分解为若干个简单数据或结构数据，每个结构数据仍

可再分。

数据类型是指一种类型，以及定义于这个值集合上的一组操作的总称。例如在高级程序设计语言中已实现了的，或非高级语言直接支持的数据结构即为数据类型。在程序设计语言中，一个变量的数据类型不仅规定了这个变量的取值范围，而且定义了这个变量可用的操作。例如，一个变量 K 定义为整型，则它可能的取值范围是－32768～32767(对于 32 位系统则为－2147483648～2147483647)，可用的操作有双目运算符＋、－、＊、/、％，单目运算符＋、－，关系运算符＜、＞、＜＝、＞＝、＝＝、!＝，赋值运算符＝等。在如 C 或 PASCAL 语言这样的程序设计语言中，不但规定了一些基本的数据类型，还提供了一些构造组合类型(如数组型、构造型、文件型等)的规则，程序员可以利用这些规则，自行定义为解决应用问题所必需的数据类型，它是确切地描述数据对象，正确地进行相关计算的有效工具。

例如，在 C 语言中对一个数据表(data list)的构造型类型定义如程序 1.1 所示。

程序 1.1　数据表(data list)的构造型类型定义

```
#define ListSize 100                    //表空间的大小，可根据实际情况决定
typedef int DataType;                   //表中元素的数据类型，假定为 int
typedef struct {
    DataType data[ListSize];            //存放表元素的向量
    int length;                         //当前的表长度
} DataList;
```

注意，数据类型的逻辑概念与其在计算机程序中的实现有很重要的区别。例如，线性表数据类型有两种传统的实现方式：**基于数组的顺序表示**和**基于链表的链接表示**。可以采用链表或数组作为线性表数据类型的存储表示。但是，“数组(array)”的概念有歧义，它既可以指一种数据类型，又可以指一种实现方式。在计算机程序设计中，“数组”常用来指一块连续的内存空间，每一个内存空间存储一个固定长度的数据项，从这个意义上讲，数组是一个存储结构。然而，数组也能够表示一种由一组结构相同的数据项组成的逻辑数据类型，每一个数据项由一个特定的索引号(即数组中的下标)来标识。从这个意义来讲，可以采用多种不同的方法来实现数组。

1.2.2　数据抽象与抽象数据类型

在软件设计时，常常提到“抽象”和“信息隐蔽”。那么，什么是抽象呢？

抽象的本质就是抽取反映问题本质的东西，忽略非本质的细节。对于数据的抽象，可以用一个例子说明。在计算机中使用二进制定点数和浮点数实现数据的存储和运算，而在汇编语言中则给出了各种数据的自然表示，如 15.5、1.3E10、10 等，它们是二进制数据的抽象，使用者在编程时可以直接使用它们，不必考虑实现的细节。到了高级语言，给出了更高一级的数据抽象，出现了整型、实型、字符型、双精度型等。待到抽象数据类型出现，可以进一步定义出更高级的数据抽象，如各种表、队列、图，甚至窗口、管理器等。这种数据抽象的层次为设计者提供了有力的手段，使得设计者可以从抽象的概念出发，从整体上进行考虑，然后自顶向下，逐步展开，最后得到所需的结果。

抽象数据类型通常是指由用户定义，用以表示应用问题的数据模型，是将数据结构作为一个软件构件的实现。抽象数据类型由基本的数据类型组成，并包括一组相关的服务(或称

操作)。抽象数据类型有些类似于 PASCAL 语言中的记录(record)类型和 C 语言中的构造(struct)类型,但它增加了相关的服务。下面给出自然数(Natural Number)的抽象数据类型定义。在 C ++ 语言中,使用关键字 struct 或 class 定义抽象数据类型。例如,自然数的抽象数据类型定义如程序 1.2 所示。

程序 1.2　自然数的抽象数据类型

```
ADT NaturalNumber IS
/ * objects: 自然数是整数的有序子集合, 它开始于 0, 结束于机器能表示的最大整数 MAXINT * /
{
    function: 对于所有的 x,y∈NaturalNumber, +、-、<、==、=等都是可用的服务。
    Zero ( ): NaturalNumber              返回 0;
    IsZero (x): Boolean                  if (x == 0) 返回 True; else 返回 False;
    Add (x,y): NaturalNumber             if (x+y <= MAXINT) 返回 x+y;
                                         else 返回 MAXINT;
    Equal (x,y): Boolean                 if (x == y) 返回 1; else 返回 0;
    Successor (x): NaturalNumber         if (x == MAXINT) 返回 x;
                                         else 返回 x+1;
    Subtract (x,y): NaturalNumber        if (x < y) 返回 0; else 返回 x-y;
} //NaturalNumber
```

对于一个其数据元素完全相同的数据类型,如果给它赋予不同的语义,即定义具有不同功能的一组,则可形成不同的抽象数据类型。例如队列和优先级队列,它们可能都是相同的顺序表结构,但其语义不同,队列是先进先出,优先级队列是优先级高的先出,具有各不相同的服务,是不同的抽象数据类型。

为了保证抽象数据类型中每一个操作的正确性,在操作中需要给出前置条件(IF)和后置条件(…THEN…ELSE)。前者给出操作正确执行所需的先决条件,后者表明在前置条件确定后应得到的结果。在 C ++ 中使用断言(assert. h)实现这种机制。如在程序中插入 assert ($x > 0$) 语句,它表明,如果满足 $x > 0$ 则可继续执行后续的程序,否则将调用通用库 stdlib. h 中的函数 abort,打印出错行号和文件名,终止程序的执行。后续章节将引用这种机制。此外,在 C ++ 中可使用"抛出异常"处理的机制 try-throw-catch。这种机制可以检测和抛出函数无法处理的异常情况。如果程序系统中有相关的异常处理器,则异常处理器捕获和处理这个异常;如果程序系统中没有相关的异常处理器,则程序终止。

抽象数据类型的特征是使用与实现分离,实行封装和信息隐蔽。就是说,在抽象数据类型设计时,把类型的声明与其实现分离开来。

首先根据问题的要求,定义该抽象数据类型需要包含哪些信息,并根据功能确定公共接口中的服务,使用者可以使用公共接口中的服务对该抽象数据类型进行操作。另一方面,抽象数据类型的物理实现作为私有部分封装在其实现模块内,使用者不能看到,也不能直接操作该类型所存储的数据,只能通过界面中的服务来访问这些数据。这样做的好处是严格区分了抽象数据类型的两个不同的视图。

从使用者的角度来看,只要了解该抽象数据类型的规格说明,就可以利用其公共接口中的服务来使用属于这个类型的数据,而不必关心其物理实现。这样使用者可以在开发过程中抓住重点,集中精力考虑如何解决应用问题,使问题得到简化。例如,在求解一个最优化

问题时常常要使用一个栈。那么，应当首先考虑此栈应存放什么信息，应如何组织，至于栈怎样实现，可能会出现哪些例外情况，这些例外情况如何处理等可以暂时忽略。待以后具体实现时再考虑。

从实现者的角度来看，把抽象数据类型的物理实现封装起来，有利于编码，测试，也有利于将来的修改。因为这样做可以使得错误局部化，一旦出现错误，其传播范围不致于影响其他模块；如果为了提高效率希望改进数据结构，可能需要改变抽象数据类型的物理实现，但只要界面中的服务的使用方式不变，其他所有使用该抽象数据类型的程序都可以不变。从而大大提高系统的稳定性。

1.3 作为 ADT 的 C ++ 类

1.3.1 面向对象的概念

什么是面向对象？Coad 和 Yourdon 给出如下的定义：

面向对象＝对象＋类＋继承＋消息通信

定义中的对象是指在应用问题中出现的各种实体、事件、规格说明等，它是由一组属性值和在这组值上的一组服务（或称操作）构成。其中，属性值确定了对象的状态。例如，对于一个显示在计算机屏幕上的矩形，它作为一个几何对象，由左上角坐标、右下角坐标、边线颜色、内部颜色等属性值确定了它的位置、颜色等状态。可以通过对象的服务来改变该对象的属性值，即对象的状态。例如，矩形可以有一个服务 move(x,y)，使用这个服务可以把矩形移到由整数 x，y 指定的新位置上。矩形还可以有两个服务 setEdgeColor(c) 和 setInterColor(c)，用以改变矩形的边和内部的着色。

在面向对象方法中，把具有相同属性和服务的对象归到同一个类(class)，而把一个类中的每一个对象称为该类的一个实例(instance)。它们具有相同的服务。例如，在计算机屏幕上有大大小小若干个矩形，分别表示各种按钮和窗口，它们都是“矩形”类的实例。这些实例共享该类的所有服务，但因其在屏幕上显示的位置、大小、颜色各不相同，所以它们的属性值也各不相同。

引入继承机制是面向对象方法的最有特色的方面。例如，在绘图系统中，可将几何图形分为点、多边形、折线、二次曲线等，它们可以各自建立相应的类，确定必需的属性和相关的操作。而多边形又可分为四边形、三角形；二次曲线又可分为椭圆、圆、抛物线、双曲线。我们以“多边形”类为基类(base class)，建立它的派生类(derived class)，如“四边形”类、“三角形”类等。将各派生类中的共同部分，包括属性和服务，集中到基类中去，派生类中只保留自己特有的属性和服务。派生类的各对象独享该派生类的属性和服务，同时还能共享基类的共有的和保护性的属性和服务。这样做的好处是可以合理地将各个对象的属性和服务分配到所有的类中，减少数据存储和程序代码的重复。通过建立继承的层次结构，还可以很方便地建立系统的体系结构，很容易地在这个结构上增加新的继承，并可有效地使用以前已经完成的类，实现类的复用。

基类又称父类、超类或泛化类，派生类又称为子类或特化类。

各个类的对象间通过消息进行通信。所谓消息，实际上是一个类的对象要求另一个类

的对象执行某个服务的指令，指明要求哪一个对象执行这个服务，必要时还要传递调用参数。系统功能的实现，就是通过一系列对象消息的传递，执行一系列服务达到的。

面向对象开发方法与传统的开发方法不同。传统的开发方法也叫做面向过程的开发方法，首先着眼于系统要实现的功能。从系统的输入和输出出发，分析系统要做哪些事情，进而考虑如何做这些事情，用自顶向下、逐步进行功能分解的方式建立系统的功能结构和相应的程序模块结构。但是，程序因各种各样的原因需要经常修改，这种修改常常涉及许多模块，有时因功能改变导致全部模块都要变更，这样修改工作量极大，很容易产生新的错误，使得程序退化。

面向对象开发方法的出现，弥补了传统方法的不足。面向对象开发方法首先着眼于应用问题所涉及的对象，包括各种物理实体、事件、规格说明，识别为解决问题所需的各种对象、对象的属性、必需的操作，以及各个对象的实例之间的关系，从而建立对象的结构和为解决问题需要执行的事件序列(俗称场景)，据此建立类的继承层次结构，通过各个类的实例之间的消息连接，实现所需的功能。类的定义充分体现了抽象数据类型的思想，基于类的体系结构可以把程序的修改局部化。特别是一旦系统功能需要改变，主要修改类中间的某些服务，类所代表的对象基本不变，整个系统保持稳定，确保系统不致因修改而退化。因此，用面向对象开发方法建立起来的软件易于修改，与传统的方法相比，程序具有更好的可靠性、可修改性、可维护性、可复用性、可适用性和可理解性。

1.3.2 C ++ 中的类

C ++ 对于面向对象程序设计的支持，核心部分就是类的定义。类的定义体现了抽象数据类型的思想，可用以支持说明与实现的分离，将抽象数据类型的实现封装在类的内部，使达到信息隐蔽的原则。为此，对类的成员来说，规定有三级存取：公共(public)、私有(private)和保护(protected)。

对于在 public 域中声明的数据成员和函数成员(也叫做成员函数)，程序中其他类的对象或操作都能请求该类的对象执行它们，因此，这些数据成员和成员函数构成类的接口部分。在 private 域和 protected 域中声明的数据成员和成员函数构成类的私有部分，只能由该类的对象和成员函数，以及被声明为友元(friend)的函数或类的对象才能访问它们。此外，在 protected 域中声明的数据成员和成员函数，还允许该类的派生类访问它们；在 private 域中声明的数据成员和成员函数，则不允许该类的派生类访问它们。

程序 1.3 给出一个 point 类的声明。在 public 域声明的函数构成这个类的公共接口，其他类的实例可以请求 Point 类的实例执行这些共有函数。例如，Point 和 get_x，以及加另一个点的命令 operator+ 都是 Point 类的用户可用的函数。在 private 域声明的成员仅允许该类的成员函数或友元函数存取。例如，Point 类中一个点的表示由两个整数实例变量 x、y 组成。类的用户不能直接访问它们，因为它们是在 private 域中声明的。类的用户只能通过 Point 类的成员函数和友元函数，包括流输入输出命令，来存取它们。

程序 1.3　Point 类的定义

```
#ifndef POINT_H
#define POINT_H                    //In the header file point.h
#include <iostream.h>
```

```
class Point{                          //类定义
public:                               //共有域
    Point(int,int);                   //构造函数
    Point(Point &);                   //复制构造函数
    ~Point();                         //析构函数
    int get_x();                      //存取函数
    int get_y();
    void put_x(int a);
    void put_y(int b);
    Point operator+(point p);         //重载函数：当前对象+参数表中对象 p
    Point operator*(int i);           //重载函数：当前对象*参数表中整数 i
    int operator>(Point p);           //重载函数：判断当前对象>参数表中对象 p 否
    int operator<(Point p);           //重载函数：判断当前对象<参数表中对象 p 否
    int operator==(Point& p);         //重载函数：判断当前对象=参数表中对象 p 否
private:
    int x;
    int y;
    friend istream& operator>>(istream& in, Point& p);    //友元函数：输入
    friend ostream& operator<<(ostream& out, Point& p); //友元函数：输出
};
#endif
```

为了存取一个点的 x、y 分量，类提供了 4 个函数 get_x()、get_y()、put_x(int a)和 put_y(int b)。之所以要用 private 关键字来保护数据的表示并提供存取函数，是为了防止类的用户直接使用数据的内部表示来编写代码，因此使用存取函数来操作数据，以维持类的抽象性。例如，若决定使用极坐标表示一个点，即使用一个角度 theta 和一个离开原点的距离 r，此时，只要类的公有函数的使用界面不改变，那么改变类的实现将不会对类的用户有任何影响，用户仍然可以通过 public 界面的存取函数使用直角坐标。

系统开发的一种方法是把类的声明放在 header 文件中，成员函数的实现分开放在代码文件中，在函数的实现代码中通过作用域设定命令“::”，将该函数归属到某一类。也可以把类的声明和成员函数的实现都放在 header 文件中。在头文件(point.h)中程序的头尾放上#ifndef POINT_H、#define POINT_H 和#endif。

Point 类的输出函数的实现代码可以为：

```
ostream& Point::operator<<(ostream& strm, Point p){
    return strm << "(" << p.get_x() << "," << p.get_y() << ")";
};
```

这个函数把点 p 的值以“x, y”的格式送到 strm 指明的输出流中去。

注意，C++ 扩充了 C 中 struct 型的功用，加进成员函数以说明一个类。在 C++ 中 struct 与 class 的区别在于：在 struct 中，默认的访问级别是 public。若在 struct 内部自始至终缺省访问级别，则所有的成员都在公共接口中。而在 class 中，默认的访问级别是 private。除此之外，struct 与 class 是等价的。

与 struct 一样，用 union 也可以定义类。在 C++ 中，union 可包含函数和变量，还可包

含构造函数和析构函数。C ++ 的 union 保留了所有 C 的特性，主要是让所有的数据成员共享相同的存储空间，在程序设计语言中称之为“等价变量”。与 class 和 struct 相比，union 可节省存储。与 struct 相似，union 中默认存取级别是 public。

1.3.3 C ++ 中的对象

1. 建立类的对象

类与对象的关系类似于语言程序中的数据类型与变量，类可以视为数据类型，它一旦定义，在整个程序运行过程中是不会变化的；而对象是在程序运行过程中建立，并在程序运行过程中撤销，它是有生命周期的。类通过建立属于该类的对象（或实例）参加运算，对象根据问题要求有其实际含义。

建立类的对象也称为实例化，采用的方式类似于定义 C 变量的方式，可以自动地，或静态地，或通过动态分配来建立。例如，建立一个 Point 类实例的语句是：

```
Point p(6,3);                    自动地
Point q;                         自动地
static Point s(3,4);             静态地
Point * t = new Point(1,1);      通过动态分配
```

对象 p、q 和 s 都是 Point 类的对象，而 t 是对象指针。

2. 构造函数

当遇到以上的每一个语句时，将隐式地调用一个构造函数（constructor），这个构造函数属于一个与它同名的类。例如，在程序 1.3 中给出的 Point 类的定义中声明了两个构造函数，构造函数的参数用于初始化对象的数据成员。如当使用声明 Point p(6,3) 建立 Point 类的对象 p 时，调用了构造函数 Point(int,int)，通过以下函数定义，将其 x，y 分量设定为 6，3：

```
Point::Point(int a,int b){x = a; y = b;}
```

或

```
Point::Point(int a,int b):x(a),y(b){}
```

这两种形式是等效的。

构造函数可以定义默认值。例如

```
Point::Point(int a = 0, int b = 0):x(a),y(b){}
```

当定义实例时给定初始值，则该实例以给定初始值来初始化其数据成员。例如，

Point p(6,3)，则用 x=a=6，y=b=3

当定义实例时未给出初始值，则该实例以默认值来初始化其数据成员，例如，

Point q，则用 x=a=0，y=b=0 来初始化其数据成员

3. 析构函数

当要撤销对象时，需要隐式地调用另一个函数，叫做析构函数（destructor），它属于名字

相同的类，但在名字前面加上了一个“～”，例如～Point()。

一个类可以定义几个构造函数，但只能定义一个析构函数。当控制要退出自动变量的作用域时，或当通过 delete 命令释放一个动态分配的变量时，就要调用析构函数。当 main 函数执行结束时，将释放静态声明的变量。

一个析构函数用于在删除一个类的对象时做清除工作。

1.3.4 C ++ 的输入输出

在 C ++ 中执行输入输出操作，需要用 # include 预处理指令包括一个＜iostream. h＞头文件。用它可支持 C ++ 的流(stream)操作。“流”是个简单的字符序列。在 C ++ 中有两个预定义的类 istream 和 ostream，它们定义了输入流和输出流。在 C ++ 语言程序中，基本的输入输出方式有两种：键盘屏幕输入输出和文件输入输出。

1. 键盘屏幕输入输出

在 C 中有用于定向到键盘输入设备、屏幕输出设备和错误文件的命令 stdin、stdout 和 stderr。在 C ++ 中用 cin、cout 和 cerr 来定义键盘输入类、屏幕输出类和错误信息输出类。操作符＜＜用于写出类 ostream 的一个对象，对于一系列输出对象，可用＜＜分开。操作符＞＞用于读入类 istream 的一个对象。

在程序 1.4 中使用了流 cin ＞＞，相继从标准输入设备上输入两个整型变量 a 和 b，并将它们打印到标准输出设备上。

程序 1.4　流操作使用示例

```
# include <iostream. h>
void main(){
    int a, b;
    cin >> a >> b;
    cout << "a: " << n << "f: " << f << endl;
}
```

在输出语句中最后输出的 endl 是 C ++ 的 I/O 操作符，它的用途是输出一个换行符并清空流。

C ++ 中的输入输出可以是自由格式，程序员不需要使用格式化符号来指定输入输出项的类型和顺序。与其他 C ++ 操作符一样，输入输出操作符能够被重载。

2. 文件输入输出

C ++ 中的文件输入输出方式如下：

(1) 在程序开头必须用预处理指令 # include 包含头文件＜fstream. h＞，它定义了类 ifstream、ofstream 和 fstream；

(2) 要创建一个输入流，必须声明它为 ifstream 类的实例；

(3) 要创建一个输出流，必须声明它为 ofstream 类的实例；

(4) 执行输入和输出操作的流必须声明它为 fstream 类的实例。

程序 1.5　在程序中使用文件的事例

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

```
#include <stdlib.h>
void main(){
    ifstream inFile;                                    //inFile 为输入流对象
    ofstream outFile;                                   //outFile 为输出流对象
    outFile.open("my.dat", ios::out);                   //建立输出文件"my.dat"
    char univ[] = "Tsinghua", name[10];
    int course = 2401, number;
    outFile << univ << endl;                            //输出到"my.dat"
    outFile << course << endl;
    inFile.open("my.dat",ios::in|ios::nocreate);        //打开输入文件"my.dat"
    if(!inFile){
        cerr << "不能打开 my.dat" << endl;
        exit(1);
    }
    char c;
    inFile >> name >> c >> number;
    outFile << "name: " << name << endl;
    outFile << "number: " << number << endl;
}
```

ifstream 类、ofstream 类和 fstream 类都是从 istream 类和 ostream 类派生出来的，而类 istream 和 ostream 又是从类 ios 派生出来的，因此这些类都可使用类 ios 的所有运算。

在调用打开文件函数 open()时，函数参数表包括实际文件名和数据流动的方向，函数返回文件的开始地址。系统在存储文件时，在其末尾添加有文件结束标记。

在程序 1.5 中，如果文件未被打开，则 outFile=0；如果文件被成功地打开，则它将代替 cout，将输出引导到文件"my.dat"中。

在文件打开的操作中，指定的文件模式有以下几种：

ios::app：把所有对文件的输出添加在文件尾。它只用于输出文件。

ios::binary：文件以二进制方式打开。此项缺省时文件以文本方式打开。

ios::nocreate：若文件不存在则将导致打开操作失败。

ios::out：表明该文件用于输出。此项可缺省。

ios::in：表明该文件用于输入。此项可缺省。

1.3.5 C++ 中的函数

1. C++ 函数的概念

在 C++ 中有两种函数：常规函数和成员函数。不论哪一种函数，其定义都包括 4 个部分：函数名、形式参数表、返回类型和函数体。函数的使用者通过函数名来调用该函数；调用时把实际参数传送给形式参数表作为数据的输入；通过函数体中的处理程序实现该函数的功能；最后得到返回值作为输出。程序 1.6 给出一个函数的例子。max 是函数名，int a 和 int b 是形式参数表，函数名前面的 int 是返回类型，在花括号内括起来的是函数体，它给出了函数操作的实现。

程序 1.6　求两个值 a 与 b 中的大值

```
int max(int a,int b){
    if(a > b) return a;
    else return b;
}
```

在 C ++ 中所有函数都有一个返回值,或者返回计算结果,或者返回执行状态。如果函数不需要返回值,可使用 void 来表示它的返回类型。函数的返回值通过函数体中的 return 语句返回。return 的作用是返回一个与返回类型相同类型的值,并中止函数的执行。

函数返回时可以通过引用方式,参看程序 1.7,此时在函数类型后加上一个“&”。

程序 1.7　使用引用的事例

```
#include <iostream.h>
char& replace(int m);
char s[80] = "Hello There";
main(){
    replace(5) = 'x'; cout << s;                    //用 x 代替 Hello 后面的空格
}
char& replace(int m){
    return s[m];
}
```

函数 replace()的返回类型说明为返回一个字符的引用类型,在函数执行时返回参数 m 指定的 s 数组元素的值。main()执行时把字符"x"送给 s[5]。

2. C ++ 中的参数传递

函数调用时传送给形参表的实参必须与形参在类型、个数、顺序上保持一致。参数传递有两种方式:一种是传值,这是默认的参数传递方式;一种是引用类型。

使用传值方式时,把实参的值传送给函数局部工作区相应的副本中,函数使用这个副本执行必要的功能。这样,函数修改的是副本的值,实参的值不变,参看程序 1.8 中的函数 squareByValue(int)。

使用引用类型方式传递参数时,需将形参声明为引用类型,即在参数名前加一个“&”。当一个实参与一个引用型形参结合时,被传递的不是实参的值,而是实参的地址,函数通过地址直接使用被引用的实参。函数执行后实参的值将发生改变。在把一个体积较大的对象作为参数传递时,使用引用方式将大大节省传递参数的时间,并可节省存储参数对象的副本空间。参看程序 1.8 中的函数 squareByReference(int&)。

程序 1.8　求平方

```
#include <iostream.h>
int squareByValue(int);
void squareByReference(int&);
int main() {
    int x = 2, z = 4;
    cout << "求平方值前 x =" << x << endl << "用 squareByValue 求平方值"
        << squareByValue(x) << endl << "求平方值后 x =" << x << endl;
        //求平方值前 x=2, squareByValue(x) = 4, 求平方值后 x=2
    cout << "求平方值前 z =" << z << endl << "用 squareByReference 求平方"
```

```
        << endl;
    squareByReference(z);
    cout << "求平方值后 z =" << z << endl;
        //求平方值前 z=4，求平方值后 z=16
}
int squareByValue(int a){return a * = a;}
void squareByReference(int& a){a * = a;}
```

当一个函数的返回值多于一个时，其中一个可由 return 语句返回，其他返回值可使用引用型参数返回。

注意，在使用传值型参数时，参数可以是常数、常量、变量或表达式；但在使用引用型参数时，参数只能是变量或对象。

一种特殊的引用调用方式叫做常值引用，其格式为 const Type& a，其中 Type 为参数的数据类型。在函数体中不能修改常值参数。

数组参数的传递情况比较特殊。数组作为形参可按传值方式声明，但实际采用引用方式传递，传递的是数组第一个元素的地址。在函数体内对形参的数组所做的任何改变都将反映到作为实参的数组中。此外，在参数表中一般按形如 int R[ ] 的形式声明，因此需要显式地声明数组的大小。

程序 1.9　递归函数：求数组 a[]前 n 个元素的和

```
int sum(int a[], int n){
    if(n > 0) return a[n-1]+sum(a,n-1);
    else return 0;
}
```

若传送的值参是一个对象（作为类的实例）时，在函数中就创建了该对象的一个副本。在创建这个副本时会调用该对象的复制构造函数。如果该类没有显式定义的复制构造函数，那么编译器会自动创建一个默认的复制构造函数，而在函数结束前要调用该副本的析构函数撤销这个副本。需要注意，如果一个类在构造函数中用 new 为指针成员分配了内存空间，并在析构函数中用 delete 进行释放，那么必须手工定义它的复制构造函数。因为编译器自动创建的复制构造函数只能够进行指针的简单复制，并不会分配新的内存空间，这样当副本析构后，母本占有的动态空间会被释放掉，造成错误。若采用引用方式传递对象，在函数中不创建该对象的副本，也不存在最后撤销副本的问题。但是，通过引用传递的是对象时，函数对对象的改变将影响调用的对象。

3. 成员函数的返回值

当成员函数的返回值为传值方式时，允许改变该对象的私有数据成员。当成员函数的返回值为常值传值方式时，需要在函数说明中加上 const 标识，使得该对象的私有成员不能被改变。当成员函数的返回值为引用方式时，该成员函数的返回值应是一个已存在变量（或对象）的别名。当该成员函数的返回值被改变时，其对应变量（或对象）的值将改变。当成员函数的返回值为常值引用方式时，其返回值与引用方式的成员函数返回值类同。但该成员函数不能改变该对象的私有成员。

程序 1.10　函数返回的事例

```
#include <iostream.h>
class Temperature{
private:
    float highTemp, lowTemp;                    //数据成员
public:
    Temperature(int hi, int lo)                 //构造函数
        { highTemp = hi; lowTemp = lo; }
    void UpdateTemp(float temp);                //传值返回
    float GetHighTemp()const;                   //常值返回
    float GetLowTemp()const;                    //常值返回
};

void Temperature::UpdateTemp(float temp){
    if (temp > highTemp) highTemp = temp;
    if (temp < lowTemp) lowTemp = temp;
}

float Temperature::GetLowTemp()const {
    return lowTemp;
}

float Temperature::GetHighTemp()const {
    return highTemp;
}
```

当成员函数返回值为常值传值方式或常值引用方式时，const 标识符一般放在最后。

4. 友元函数

在类的声明中可使用保留字 friend 定义友元函数。友元函数实际上并不是这个类的成员函数，它可以是一个常规函数，也可以是另一个类的成员函数。如果想通过这种函数存取类的私有成员和保护成员，则必须在类的声明中给出函数的原型，并在该函数原型前面加上一个 friend。

参看程序 1.3 给出的 Point 类的声明，有两个重载操作符“与”，因为它们直接在对象外部的程序中使用，都被声明为 Point 类的友元函数：

```
friend istream& operator>>(istream&,Point&);    //输入友元函数
friend ostream& operator<<(ostream&,Point&);    //输出友元函数
```

1.3.6 动态存储分配

在 C 语言程序中使用函数 malloc 动态地为程序变量分配它所需要的空间，并通过函数 free 动态地释放这个空间。函数 malloc 执行时，要求它的调用者使用函数 sizeof 提供所需存储空间的数量，完成动态分配后还需要对返回指针做类型的强制转换。

而 C++ 则为动态存储分配提供了两个新的命令：new 和 delete，增强了动态分配的功能。它们操纵属于可利用空间的存储并取代了 C 中的库函数 malloc 和 free。操作 new 要求以被建立对象的类型作为参数，并返回一个指向新分配空间的指针。此时，返回指针自动

根据类型说明进行了类型转换。例如，可以为分配一个整数或一个点编写如下语句：

```
int * ip = new int;
```

或

```
Point * p = new Point;
```

在 C ++ 中没用无用单元收集，这样使用 new 分配的存储必须显式地使用 delete 释放。delete 函数不需要明确指出 new 分配了多少存储。例如，如果我们建立下列有 100 个点的数组：

```
Point * p = new Point[100];
```

则通过以下命令释放该存储：

```
delete []p;
```

若释放时遗漏了“[]”，则将只释放 p 所指示的第一个元素，我们将会“失去”其他 99 个点所占据的空间，以致不能再复用它们。另一方面，若使用表达式计算出来的值超出括号中的 100，则程序将会出错且结果不可预测。因此，使用 new 和 delete 管理存储时必须仔细。

1.3.7 C ++ 中的继承

继承性(inheritance)是渐增式地修改已有的类定义以产生新类的技术。继承结构的例子如程序 1.11 所示。程序 1.11(a)中声明的类 Polygon(多边形)是程序 1.11(b)中所声明的 Quadrilateral(四边形)类的基类。类 Quadrilateral 首部的短语“:public Polygon”指明类 Quadrilateral 是类 Polygon 的子类，而且类 Polygon 的公有成员应是类 Quadrilateral 的公有成员。在成员函数 Polygon::draw 的声明中记号“= 0”指明这个方法的定义应由子类提供，记号“= 0”还使得 Polygon 成为一个抽象类。这意味着 Polygon 类没有实例，因而所有操作都没有实现代码。

程序 1.11　类的继承关系举例

```
class Polygon {
public:
    Polygon(Point);
    void move(Point);
    void isInside(Point);
    Point getReferencePoint();
    virtual void draw() = 0;
private:
    Point referencePoint;
};
```

(a) Polygon 类的定义

```
class Quadrilateral: public Polygon {
public:
    Quadrilateral(Point, Point);
    void isInside(Point);
    void draw();
private:
    Point vertex2;
};
```

(b) 类 Polygon 的子类 Quadrilateral 的定义

程序 1.12 展示了类 Quadrilateral 的完整定义，是复合了程序 1.11 所定义的 Quadrilateral 类和 Polygon 类而得到的结果。

注意，Polygon 类的私有成员 referencePoint 并未成为 Quadrilateral 类的私有成员，只

能使用在 Polygon 类中定义的成员函数 getReferencePoint( ) 来进行存取。因此，Quadrilateral 对象的第一个顶点(即 referencePoint)不能在 Quadrilateral 类中直接存取。

程序 1.12　Quadrilateral 类的实际定义

```
class Quadrilateral {
public:
    Quadrilateral(Point, Point);
    Point getReferencePoint();              //从 Polygon 类继承的属性
    void isInside(Point);
    void move(Point);                       //从 Polygon 类继承的属性
    void draw();
private:
    Point vertex2;
};
```

现在改变 Polygon 的声明，以使其原来被封装的数据成员对它的子类有效。程序 1.13 改变了 Quadrilateral 类继承 Polygon 类的方式。使用保留字 protected 代替程序 1.11(a) 中的 private，则 Polygon 类的子类就可以继承那些被声明为 protected 的成员。Quadrilateral 类的成员函数现在就能直接存取在程序 1.13 中给定的 referencePoint 了。

程序 1.13　基类(超类)中被保护成员的影响

```
class Quadrilateral {
public:
    Quadrilateral(Point, Point);
    Point referencePoint();                 //从 Polygon 类继承的操作
    void isInside(Point);
    void move(Point);                       //从 Polygon 类继承的操作
    void draw();
protected:
    Point referencePoint;                   //从 Polygon 类继承的属性
    Point vertex2;
};
```

1.3.8　多态性

多态性(polymorphism)是指允许同一个函数(或操作符)有不同的版本，对于不同的对象执行不同的版本。C ++ 支持以下两种多态性：

(1) 编译时的多态性，表现为函数名(或操作符)的重载；

(2) 运行时的多态性，通过派生类和虚函数来实现。

1. C ++ 中的函数名重载

函数名重载允许 C ++ 程序中多个函数取相同的函数名，但其形参或返回类型可以不同。例如，C 标准函数库中有 3 个标准函数 abs( )、labs( )和 fabs( )，分别计算整型数、长整型数和双精度型数的绝对值。在 C 中因处理的数据类型不同，必须取不同的函数名。在 C ++ 中，可以把这 3 个函数都命名为 abs( )：

```
int abs(int);
long abs(long);
double abs(double);
```

编译器能够比较具有同名的函数的特征，通过识别实参的数目和每个实参的类型，来标识使用于一个特定调用的是哪一个版本的 abs()。

2. C++ 的操作符重载

C++ 提供了一种能力，可用同一个名字定义多个操作，这种能力叫做操作符重载。例如，可以命名一个函数：clear(int *)，它将一个整数清零。还可以再命名另一个函数 clear(int[])，它把一个整数数组清零。

在 C 中，必须使用名字 clearIntArray()和 clearInt()来区分这两个函数。在 C++ 中，编译器能够比较同名函数的特征，通过识别实参的数目和每个实参的类型，来标识一个特定调用中用的是哪一个版本的 clear。

为了支持面向对象，C++ 提供了双目重载操作符，如'+'和'<'。这种操作可使得程序更可读、写得更自然。例如，可定义"点(Point)"的运算(作为成员函数)，如：

p1+p2：把两个点 (x1, y1) 和 (x2, y2) 相加成一个点 (x1+x2, y1+y2)。

p1<p2：两个点 p1 和 p2 的"小于"关系，表示 p1 比 p2 更靠近原点 (0, 0)。

p1/i：一个点 p=(x, y) 除以一个整数 i 的除法 (x/i, y/i)。

可以按以下方式说明重载操作：

```
Point operator+(const Point& p);
Point operator/(int i);
int operator<(const Point& p);
```

使用这些新的操作的表达式如：

```
Point midPoint = (point1+point2)/2;
```

或

```
if(midPoint < referencePoint) ...
```

注意：每一个这样的操作符在调用时可看成是该操作符左边对象的成员函数。例如，point1+point2 实际上是一个消息。由类 Point 的实例 point1 调用成员函数"+"，该对象的属性确定第一个操作数的值；函数参数表中指定的 Point 的实例 point2 的属性确定第二操作数的值。这种重载能力允许像使用内建类型 (如 int,float) 那样来使用用户自定义类型。与在不允许重载操作的语言中相同的语句比，这样可以改善程序的可读性。

但重载作为非成员函数的(双目)操作符时，参加运算的两个操作数必须都出现在参数表中。第一个参数是第一操作数，第二个参数是第二操作数。例如，若用 struct 定义 worker 记录结构如下：

```
struct worker{
    int id;
    char name[20];
    float age;
```