第 1 章　绪论

教学要点

数据结构与算法是一门讨论如何在计算机中构建描述现实世界实体的计算模型并实现其操作的学科。数据的表示组织和处理方法直接关系到软件的运行效率，所以软件设计时要考虑的首要问题是数据的表示、组织和处理方法。数据结构与算法课程不仅是一般程序设计的基础，而且是设计和实现系统程序和应用程序的重要基础。

本章是整个课程的绪论，将讨论数据结构与算法分析中重要的基本概念，如数据、抽象数据类型、数据结构、算法等，以及算法分析的基本方法。此外还将介绍 C#程序设计语言最基本的内容。

建议本章授课 4-12 学时，其中数据结构与算法的基本概念部分 3 学时，C#语言 1-9 学时。

1.1　数据结构的基本概念

随着计算机产业的飞速发展，计算机的应用范围迅速扩展，计算机已深入到人类社会的各个领域，计算机技术日益显现其重要作用。软件设计是计算机科学与技术各个领域的主体任务之一，而数据结构设计和算法设计是软件系统设计的核心。在计算机领域流传着一句源于著名计算机科学家 Niklaus Wirth 教授的经典名言，就是“数据结构+算法=程序”。这个“公式”简洁明了地指明了程序和数据结构与算法的关系，因而也说明了数据结构和算法课程的重要性。

1.1.1　数据类型与数据结构

1. 数据、数据项和数据元素

数据（data）是计算机程序的处理对象，包括描述客观事物数量特征的数值数据以及名称特性的字符数据等，也就是说，数据是以多种形式呈现的信息，可以是任何能输入到计算机并等待其加工处理的符号集合的总称。例如，学生信息管理系统所处理的数据是一所学校每个学生的信息，包括学号、姓名、年龄和各科成绩等；科研设备管理系统处理的数据是每台设备的信息，包括设备号、设备类型、名称和保管人等；图像和视频处理软件接受和处理的数据是经过专业设备采集并数字化的图像和视频信号。随着技术的进步，数据的形式也越来越多。

数据的基本单位是数据元素（data element），它是表示一个事物的一组数据，通常作为一个整体进行考量和处理，有时又称为数据结点。在很多问题中，一个数据元素可能分成若干成分，构成数据元素的某个成分的数据称作该数据元素的数据项（data item），有时又称为数据域（data field），数据项是数据元素的基本组成单位。

2. 数据类型

在用高级程序语言（如 C 语言和 C#语言）编写的程序中，必须对程序中出现的每个变量、常量

或表达式，明确说明它们所属的数据类型。数据类型（data type）如同一个模板，它定义了属于该种类型的数据的性质、取值范围以及对该数据所能进行的各种操作。例如，C#语言中整数类型 int 的值域是{-2^{31}，…，-2，-1，0，1，2，…，$2^{31}-1$}，对这些值进行的操作包括加减乘除、求模、相等或不等比较操作或运算等等。

高级程序设计语言都提供了一些基本数据类型，如 C#语言中有 int、long、float、double、char、string 等基本数据类型。这些基本数据类型在数据处理程序中应用得最为频繁，但是它们往往不能满足程序设计中的所有需求，这时可以利用基本类型设计出各种复杂的数据类型，称为自定义数据类型。自定义数据类型要声明一个“值”的集合和定义在此集合上的 “一组操作”。例如可以定义“学生”类型，它是一种复合类型，包括学号、姓名和成绩等信息，学生姓名可以用字符串类型 string 表示，年龄可以用整数类型 int 表示，成绩则可以用浮点类型 float 表示等。

3. 抽象数据类型

为了描述更广泛范围的数据实体，数据结构和算法描述中使用的数据类型不仅仅局限于程序设计语言中的数据类型，而更多地是指某种抽象数据类型（Abstract Data Type，ADT）。抽象数据类型是指一个概念意义上的类型和这个类型上的逻辑操作集合。

相对于编程语言中的数据类型，抽象数据类型的范畴更为广泛。一般地说，数据类型指的是高级程序设计语言支持的数据类型，包括固有数据类型和自定义数据类型；而抽象数据类型是数据与算法在较高层次的描述中用到的概念，指的是在常规数据类型支持下用户新设计的高层次数据类型。

抽象数据类型具有“数据抽象” 和“数据封装”两个重要特征。数据抽象特征表现在：用抽象数据类型描述程序处理的实体时，强调的是数据的本质特征、其所能完成的功能以及它和外部的接口（即外界使用它的方法）。数据封装特征表现在：抽象数据类型将实体的外部特性和其内部实现细节分离，并且对外部用户隐藏其内部实现细节。

数据类型和抽象数据类型实质上是相互关联的，有时甚至是等价的。我们将要讨论线性表、栈、队列、串、数组、树和二叉树、图等典型的数据结构，在描述它们时，这些典型的数据结构就是一个个不同的抽象数据类型，在实现它们时，我们就定义了相应的数据类型。

4. 数据结构

计算机处理的数据一般很多，但它们不是杂乱无章的，众多的数据间往往存在着内在的联系，对大量的、复杂的数据进行有效处理的前提是分析清楚它们的内在联系。

数据结构（data structure）是指数据元素之间存在某种关系的数据集合。例如，一个按设备号排列的科研设备信息的数据集合（科研设备信息表），就是一个具有“顺序”关系的数据结构，这种关系不因数据的改变而改变。

数据结构可以看成是关于数据集合的数据类型，它关注三个方面的内容：数据元素的特性、数据元素之间的关系以及由这些数据元素组成的数据集合所允许进行的操作。例如，前面提到的科研设备信息表具有顺序关系，可以增加新的设备信息或删除已有设备的信息；由祖父、父亲、我、儿子、孙子等成员组成的家族数据结构显然具有层次关系，可以增加新的成员或计算某成员所处的层次。

数据结构课程主要讨论三方面的问题：数据的逻辑结构、数据的存储结构和数据的操作。后面将陆续介绍相关的概念。

【例1.1】 用 C#语言描述学生信息和学生信息表数据结构。

假设要描述的学生信息包括学生的学号、姓名、性别、年龄等数据。每个学生的相关信息一起构成学生信息表中的一个数据元素，其中学号、姓名、性别、年龄等数据就构成学生情况描述的数据项。表 1.1 是一个有 3 个数据元素的学生信息表。

学生信息可以用 C#语言声明为如下的类型：

```
public class Student{
    public string studentID;
    public string name;
    public string gender;
    public int age;
}
```

表 1.1 学生信息表

学号	姓名	性别	年龄
200518001	王兵	男	18
200518002	李霞	女	19
200518003	张飞	男	19

学生信息表则是由 Student 类型的数据元素组成的、能够进行特定操作的数据集合，即学生信息表是一种特定类型的数据结构。学生信息表可用 C#语言如下定义：

```
class StudentInfoTable{
    Student[] studentList;                  //学生信息表内部存储数组（块）
    public int Add(Student st);             //将新学生添加到表的结尾处
    public bool Contains(Student st);       //确定某个学生是否在表中
    public void Sort();                     //对表中元素进行排序
}
```

1.1.2　数据的逻辑结构

数据的逻辑结构侧重于数据集合的抽象特性，它描述数据集合中数据元素之间的逻辑关系。一般可用一个数据元素的集合和定义在此集合上的若干关系来表示数据元素之间的逻辑关系，即数据的逻辑结构。数据结构这一术语很多时候指的就是数据的逻辑结构。

按照数据集合中数据元素之间存在的不同特性的逻辑关系，常见的数据结构可以分为三种基本类型：线性结构、树结构和图结构。

1. 线性结构

线性数据结构是一组具有某种共性的数据元素按照某种逻辑上的顺序关系组成的一个数据集合。线性结构具有的特性是：数据集合的第一个数据元素没有前驱数据元素，最后一个数据元素没有后继数据元素，其他的每个元素只有一个前驱元素和一个后继元素。

线性结构如图 1.1 (a)所示，其中数据元素 *B* 有一个前驱数据元素 *A*，有一个后继数据元素 *C*，*A* 是该数据集合中的首数据元素，没有前驱数据元素，*C* 是尾数据元素，没有后继数据元素。

2. 树结构

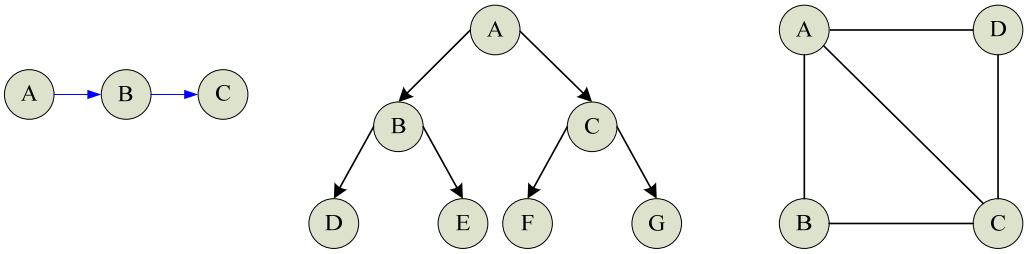
树状数据结构是一组具有某种共性的数据元素按照某种逻辑上的层次关系组成的一个数据集合。树结构具有的特性是：数据集合有一个特殊的数据元素称为根（root）结点，它没有前驱数据元素；树中其他的每个数据元素都只有一个前驱数据元素，可有零个或若干个后继数据元素。

树结构如图 1.1 (b)所示，其中数据元素 *A* 是根结点，它有两个后继数据元素 *B* 和 *C*，没有前驱数据元素；数据元素 *B* 有一个前驱数据元素 *A*，有两个后继数据元素 *D* 和 *E*。

3. 图结构

图状数据结构是一组具有某种共性的数据元素按照某种逻辑上的网状关系组成的一个数据集合。图结构具有的特性是：数据集合的每个数据元素可有零个或若干个前驱数据元素，可有零个或若干个后继数据元素，即每个数据元素可与其他零个或若干个元素有关系。

图结构如图 1.1 (c)所示，其中数据元素 *C* 有两个前驱数据元素 *A* 和 *B*。



(a) 线性结构　　(b) 树结构　　(c) 图结构

图 1.1 三种基本的数据结构

经常用图 1.1 所示的图示法表示数据的逻辑结构。在图示法中，圆圈表示一个数据元素（数据结点），圆圈中的字符或数字表示数据元素的标记或数据元素的值，连线表示数据元素间的逻辑关系。

1.1.3 数据的存储结构

数据的逻辑结构是软件设计人员从逻辑关系的角度观察和描述数据，而为了在计算机中实现对数据的操作，还需要按某种方式在计算机中表示和存储这些数据。数据集合在计算机中的存储表示方式称为数据的存储结构，也称为物理结构。

数据的逻辑结构具有独立于计算机的抽象特性，数据的存储结构则依赖于计算机，它是逻辑结构在计算机中的实现。

1. 顺序存储结构和链式存储结构

基本的数据存储结构有两种形式：顺序存储结构和链式存储结构。

顺序存储结构将数据集合中的数据元素存储在一块地址连续的内存空间中，并且逻辑上相邻的元素在物理上也相邻。例如，用 C 语言中的数组可以实现顺序存储结构，数组元素之间的顺序体现了线性结构中元素之间的逻辑次序，数据元素的存储位置由其在集合中的逻辑位置确定。

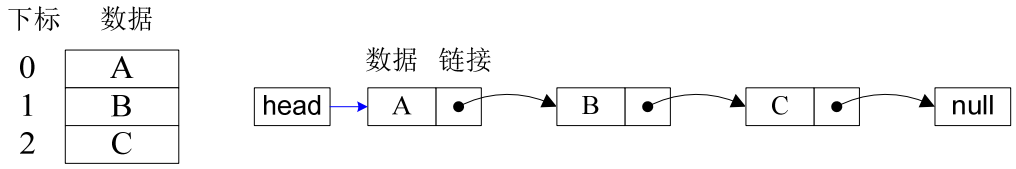
链式存储结构使用称为结点（node）的扩展类型存储各个数据元素，结点由数据元素域和指向其他结点的指针域组成，链式存储结构使用指针将相互关联的结点链接起来。数据集合中逻辑上相邻的元素在物理上不一定相邻，元素间的逻辑关系表现在结点的链接关系上。

在顺序存储结构中，所有分配的存储空间都被数据元素自身占用了；而在链式存储结构中，每个结点至少由两部分组成：数据域和指针域，数据域保存数据元素的数据，指针域指向相关结点。指针域保存指向相关结点的链信息，因此又称为链域。

顺序存储结构和链式存储结构是两种常用的基本存储结构。用不同方式组合这两种基本存储结构，可以产生复杂的存储结构。

【例1.2】 线性表的两种存储结构。

对于数据元素是{A，B，C}的线性结构，其顺序和链式存储结构如图 1.2 所示。



(a) 顺序存储结构　　(b) 链式存储结构

图 1.2 两种不同的存储结构

2. 存储密度

一个数据结构所需的存储空间不仅用来存放数据本身，也可能存放其他的信息。数据结构的存储密度定义为数据本身所占用的存储量和整个数据结构所占的存储量的比值，即：

$$\text{存储密度} = \frac{\text{数据本身所占的存储量}}{\text{整个结构所占的存储总量}}$$

如果数据结构所有的存储空间都用来存储数据元素，则这种存储结构是紧凑结构，紧凑结构的存储密度为 1，顺序存储结构是紧凑结构。

如果数据结构的所有存储空间不仅用来存储数据本身，也存储其他的信息，则这种存储结构是非紧凑结构，它的存储密度小于 1。存储密度越大，则存储空间的利用率越高；存储密度低，说明附加的信息可能较多，占用的存储空间大，但这可能会带来操作上的便利。链式存储结构是非紧凑结构。

3. 数据存储结构的选择

计算机运行任何程序都要花费一定的时间和占用一定的内存空间，理想的情况是花费的时间少和占用的空间小，但有时时间和空间的要求相互矛盾，所以在软件设计时，除了用正确的逻辑结构描述要解决的问题外，还应选择一种合适的存储结构，使得所实现的程序在以下两方面的综合性能最佳：数据操作所花费的时间和程序所占用的存储空间。

例如，对于线性表的存储，可以按下面两种情况分别处理：

1) 当不需要频繁插入和删除时，可以采用顺序存储结构，此时占用的存储空间少。
2) 当插入和删除操作很频繁时，需要采用链式存储结构。此时虽然占用的存储空间较多，但操作的时间效率高。这种方案以存储空间为代价换取了时间效率。

1.1.4 数据的操作

在数据结构中数据的操作指的是对数据集合对象所能进行的某种处理，对一个数据结构进行的所有操作构成该数据结构的操作集合。

每种特定的逻辑结构都有一个自身的操作集合，不同的逻辑结构有不同的操作集合。例如对于一个线性结构，尽管它的存储结构可能有多种方式，在该线性数据集合上可以定义以下几种常用的操作：

- 获取或设置数据集合中某元素的值。
- 统计数据集合的元素个数。
- 插入新的数据元素。
- 删除某数据元素。
- 在数据结构中查找满足一定条件的数据元素。
- 将数据集合的元素按某种指定的顺序重新排列。

数据的操作是定义在数据的逻辑结构上的，但操作的具体实现则与数据的存储结构有关。例如对

于一个线性表，选择顺序存储结构还是链式存储结构对于插入或删除操作，都会造成不同的实现方式。

1.2 算法与算法分析

1.2.1 算法

算法（Algorithm）是对特定问题求解过程的一种描述，它定义了解决该问题的一个确定的、有限长的操作序列。

1. 算法定义

数据结构与算法领域的经典著作《The Art of Computer Programming》的作者、图灵奖获得者、著名计算科学家 D.Knuth 对算法做过一个为学术界广泛接受的描述性的定义：算法是一个有穷规则的集合，其规则确定了一个解决某一特定类型问题的操作序列。算法的规则具有如下 5 个重要特征：

- 确定性 — 对于每种情况下所应执行的操作，在算法中都有确切的规定，算法的执行者或阅读者都能明确其含义，并且在任何条件下，算法都只有一条执行路径。
- 可行性 — 算法中的所有操作都必须是足够基本的，都可以通过已经实现的基本操作运算有限次予以实现。
- 有穷性 — 对于任意一组合法输入值，算法必须在执行有穷步骤之后结束。算法中的每个步骤都能在有限时间内完成。
- 有输入 — 算法有零个或多个输入数据，即算法的加工对象。有些输入量需要在算法执行过程中输入，而有的算法表面上可以没有输入，实际上输入量已被嵌入算法之中。
- 有输出 — 算法有一个或多个输出数据，它是算法进行信息加工后得到的结果，与“输入”有确定关系，这种关系体现算法的功能。

2. 算法的描述

算法可用文字、流程图、高级程序设计语言或类同于高级程序设计语言的伪码描述。无论哪种描述形式，都要体现出算法是由语义明确的操作步骤组成的有限序列，它精确地指出怎样从给定的输入信息得到要求的输出信息，算法的执行者或阅读者都能明确其含义。

【例1.3】 线性表的顺序查找（sequential search）算法。

在线性表中，按关键字进行顺序查找的算法思路为：对于给定值 k，从线性表的一端开始，依次与每个元素的关键字进行比较，如果存在关键字与 k 相同的数据元素，则查找成功；否则查找不成功。

顺序表的顺序查找过程如图 1.3 所示。

图 1.3 顺序存储线性表的顺序查找过程

3. 算法与数据结构

数据的逻辑结构、存储结构以及对数据所进行的操作三者是相互依存的。在研究一种数据结构时，总是离不开研究对这种数据结构所能进行的各种操作，因为，这些操作从不同角度体现了这种数据结构的某种性质，只有通过研究这些操作的算法，才能更清楚地理解这种数据结构的性质。反之，每种算法都是建立在特定的数据结构上的。数据结构和算法之间存在着本质的联系，失去一方，另一方就没有意义。

（1）同样的逻辑结构因为存储结构的不同而采用不同的算法。

线性表可以用顺序存储结构或链式存储结构实现，不同存储结构的线性表上的排序算法是不同的。例如，冒泡排序、折半插入排序等算法适用于顺序存储结构的线性表；适用于链式存储结构线性表的排序算法有直接插入排序、简单选择排序等（详见第九章“排序算法”）。

（2）同样的逻辑结构和存储结构，因为要解决问题的要求不同而采用不同的算法。

【例1.4】 大规模线性表的分块查找（blocking search）算法

在前面的例子中介绍的顺序查找算法适合于数据量较小的线性表，如学生成绩表。一部按字母顺序排序的字典也是一个顺序存储的线性表，具有与学生成绩表相同的逻辑结构和存储结构，但数据量较大，采用顺序查找算法的效率会很低，此时可以采用分块查找算法。

一部字典是按词条的字母顺序排好序的线性表，它也可以看成是由首字母相同、大小不等的若干块（block）所组成的，为使查找方便，每部字典都设计了一个索引表，指出每个字母对应单词的起始页码。

字典分块查找算法的基本思想：将所有单词排序后存放在数组 dict 中，并为字典设计一个索引表 index，index 的每个数据元素由两部分组成：首字母和下标，它们分别对应于单词的首字母和以该字母为首字母的单词在 dict 数组中的起始下标。

这样，通过索引表 index，将较长的单词表 dict 划分成若干个数据块，以首字母相同的若干单词构成一个数据块，因此每个数据块的大小不等，每块的起始下标由 index 中对应“首字母”列的“下标”标明。

使用分块查找算法，在字典 dict 中查找给定的单词 token，必须分两步进行：

- 根据 token 的首字母，查找索引表 index，确定 token 应该在 dict 中的哪一块。
- 在相应数据块中，使用顺序查找算法查找 token，得到查找成功与否的信息。

1.2.2 算法设计的要求

一个好的算法设计应达到以下目标：

- 正确性（Correctness）— 算法应确切地满足具体问题的需求，这是算法设计的基本目标。对算法是否“正确”的理解可以有四个层次：1）不含语法错误；2）对于某几组输入数据能够得出满足要求的结果；3）程序对于精心选择的、典型、苛刻且带有刁难性的几组输入数据能够得出满足要求的结果；4）程序对于一切合法的输入数据都能得出满足要求的结果；
- 可读性（Readability）— 算法既是为了计算机执行，也是为了人的阅读与交流。算法的描述应有利于人们的理解，这既有利于程序的调试和维护，也有利于算法的交流和移植；相反，晦涩难读的程序易于隐藏较多错误而难以调试。算法的可读性主要体现在两方面：一是被描述算法中的类名、对象名、方法名等的命名要见名知意；二是要有足够多的注释。
- 健壮性（Robustness）— 当输入非法数据时，算法要能做出适当的处理，而不应产生不可预料的结果。一般地，处理出错的方法不应是中断程序的执行，而应是返回一个表示错误或错误性质的值，以便在更高的抽象层次上进行处理。
- 高效性（Efficiency）— 算法的执行时间应满足问题的需求，执行时间短的算法称为高时间效率的算法；算法在执行时一般要求额外的内存空间，内存要求低的算法称为高空间效率的算法。算法应满足高时间效率与低存储量需求的目标，对于同一个问题，如果有多个算法可供选择，应尽可能选择执行时间短和内存要求低的算法。但算法的高时间效率和高空间效率通常是矛盾的，在很多情况下，首先考虑算法的时间效率目标。

1.2.3 算法效率分析

1．算法的时间复杂度

由算法编写的程序运行所需的时间，既依赖于算法本身，也取决于算法处理的数据规模，还与计算机系统的软件、硬件等环境因素有关。一个算法由控制结构和原操作构成，算法的执行时间等于所有语句执行时间的总和，它取决于控制结构和原操作两者的综合效果。为了便于比较同一问题的不同算法，通常选取一种对于所研究的问题来说是基本操作的原操作，以该基本操作重复执行的次数作为算法的某种时间度量。

算法重复执行原操作的次数是该算法所处理的数据个数 n 的某种函数 $f(n)$，其渐进特性称作该算法的时间复杂度（time complexity），记作 $T(n)=O(f(n))$，它表示随着问题规模的增大算法执行时间的增长率和函数 $f(n)$的增长率相同，通常用算法的时间复杂度来表示算法的时间效率。

$O(1)$表示算法执行时间是一个常数，不依赖于 n；$O(n)$表示算法执行时间与 n 成正比，是线性关系，$O(n^2)$、$O(n^3)$、$O(2^n)$分别称为平方阶、立方阶和指数阶；$O(\log_2 n)$为对数阶。若两个算法的执行时间分别为 $O(1)$和 $O(n)$，当 n 充分大时，显然 $O(1)$的执行时间要少。同样，$O(n^2)$和 $O(n\log_2 n)$相比较，当 n 充分大时，因 $\log_2 n$ 的值远比 n 小，则 $O(n\log_2 n)$所对应的算法速度要快得多。

时间复杂度随 n 变化情况的比较如表 1.2 所示。

表 1.2 不同的时间复杂度随 n 变化情况举例

时间复杂度	$n=8(2^3)$	$N=10$	$n=100$	$n=1000$
$O(1)$	1	1	1	1
$O(\log_2 n)$	3	3.322	6.644	9.966
$O(n)$	8	10	100	1000
$O(n\log_2 n)$	24	33.22	664.4	9966
$O(n^2)$	64	100	10 000	10^6

【例1.5】 分析算法片段的时间复杂度。

(1) 时间复杂度为 O(1)的简单语句。

```
s = 10;
```

该语句的执行时间是一常量，时间复杂度为 O(1)。

(2) 时间复杂度为 O(n)的单重循环。

```
int n = 100, sum = 0;
for(int i=0;i<n;i++) sum += a[i];
```

该 for 语句循环体内语句的执行时间是一常量，共循环执行 n 次，所以该循环的时间复杂度为 O(n)。

(3) 时间复杂度为 O(n^2)的二重循环。

```
int n = 100;
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        Console.Write(i*j);
```

外层循环执行 n 次，每执行一次外层循环时，内层循环执行 n 次。所以，二重循环中的循环体语句被执行 $n\times n$ 次，时间复杂度为 O(n^2)。如果代码改为：

```
int n = 100;
for(int i=0;i<n;i++)
    for(int j=0;j<i;j++)
        Console.Write(i*j);
```

外层循环执行 n 次，每执行一次外层循环时，内层循环执行 i 次。此时，二重循环的执行次数为 $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$，则时间复杂度仍为 O($n^2$)。

(4) 时间复杂度为 O($n\log_2 n$)的二重循环。

```
int n = 64;
for(int i=1;i<=n;i*=2)
    for(int j=1;j<=n;j++)
        Console.Write(i*j);
```

外层循环每执行一次，i 就乘以 2，直至 $i>n$，所以外层循环共执行 $\log_2 n$ 次。内层循环执行次数恒为 n。此时，总的循环次数为 $\sum_{i=1}^{\log_2 n} n = O(n\log_2 n)$，则时间复杂度为 O($n\log_2 n$)。

(5) 时间复杂度为 O(n)的二重循环。

```
int n = 64;
for(int i=1;i<=n;i*=2)
    for(int j=1;j<=i;j++)
        Console.Write(i*j);
```

外层循环执行 $\log_2 n$ 次。内层循环执行 i 次，随着外层循环的增长而成倍递增。此时，总的循环次数为 $\sum_{i=1}^{\log_2 n} 2^i = O(n)$，则时间复杂度为 O($n$)。

2．算法的空间复杂度

算法的执行除了需要存储空间来寄存本身所用指令、变量和输入数据外，也需要一些对数据进行

操作的工作单元和存储一些为实现算法所需的辅助空间。与算法的时间复杂度概念类似，算法的空间复杂度（space complexity）主要着眼于算法所需辅助内存空间与待处理的数据量之间的关系，也用O($f(n)$)的形式表示。例如，分析某个排序算法的空间复杂度，就是要确定该算法执行中，所需附加的内存空间与待排序数据序列的长度之间的关系。在冒泡排序过程中，需要一个辅助存储空间来交换两个数据元素，这与序列的长度无关，故冒泡排序算法的空间复杂度为 $O(1)$ 。归并排序算法在运行过程中需要与存储原数据序列的空间相同大小的辅助空间，所以它的空间复杂度为 $O(n)$。

1.3 C#语言简介

Microsoft 公司推出的 C# (读作 C sharp) 语言是一种新的编程语言，它是为开发运行在.NET Framework 平台上的、广泛的企业级应用程序而设计的。在面向对象的编程语言中，C#具有精确、简单、类型安全、面向对象、跨平台互用的特点；用 C#开发的应用软件在可移植性、健壮性、安全性等方面大大优于已存在的其他编程语言。.NET Framework 为软件开发提供了丰富的类库，利用这个庞大的类库，可进行面向对象的事件描述、处理和综合，极大地方便了众多领域的应用开发。

C#语言从 C 和 C++语言演化而来。它在语句、表达式和运算符方面借用了 C 和 C++中一些已有的元素，并加入新特性。C#语言在类型安全性、版本转换、事件和垃圾回收等方面进行了相当大的改进和创新。

C#语言对于数据结构和算法的描述也带来极大的便利。在以 Pascal 和 C 为代表的结构化程序设计语言中，数据的描述和对数据的操作两者是分离的，数据的描述用数据类型表示，对数据的操作则用过程或函数表示。例如，在描述字符串时，先定义字符串的数据表示，再用过程实现对字符串的操作。这是典型的面向过程的程序设计方式，用这种方式所设计的代码往往具有重用性差、可移植性差、数据维护困难等缺点。针对这个问题逐渐发展出了面向对象的程序设计思想。面向对象技术具有抽象、信息隐藏和封装、继承和多态等特性。

数据结构的三个要素，即数据的逻辑结构、存储结构以及对数据所进行的操作，实际上是相互依存、互为一体的，所以用封装、继承和多态等面向对象的特性能够更深入地刻画数据结构。例如，用 String 类来描述字符串，而串连接、串比较等操作则声明为该类的方法，String 类的设计者用面向对象思想设计这个类并实现其中的方法。关于数据的描述和对数据的操作都封装在同一个以类为单位的模块中，因此增强了代码的重用性、可移植性，使数据易于维护。String 类的使用者，只需要知道该类对外的接口，即类中的公共方法和属性，即可方便地使用字符串这样一种数据结构。

C#提供了对象的自引用方式来实现数据的链式存储结构，这种方式避免直接使用指针所带来的安全隐患，使 C#语言可以以面向对象的方式实现各种复杂的数据结构。

1.3.1 C#的安装、编辑、编译和运行

为进行 C#程序设计，可以使用多种方法。最简单的方法是使用文本编辑器（如 Windows 自带的 Notepad）和 C# 命令行编译器 (csc.exe，它包含在.NET 软件开发工具包 SDK 中) 来构建.NET 程序。.NET SDK 可以从 Microsoft 公司的网站免费下载，但在 SDK 中不包含代码生成实用工具（向导）、图形用户接口等功能的集成开发环境（IDE）。

为了帮助减轻在命令行构建软件的负担，许多.NET 开发人员都利用可视化工具，例如 Microsoft 功能齐全的 Visual Studio 2010/2015。这个软件产品的功能非常强大，不过其企业版的价格过高。除非是专业的软件工程师，否则 Visual Studio 2010/2015 所提供的大量选项通常会使学习过程复杂化。

Visual Studio 2010/2015 系列中的 Express Edition（速成版本）扩展了 Visual Studio 的产品线，它

是一种能迅速上手、轻量级易于使用的工具，对于编程爱好者和大学生来说是很好的开发 Windows 应用程序和网站的工具。C#在 Visual Studio 套件中作为 Visual C#引入，Visual C# 2010/2015 Express Edition 完全满足学习和使用 C#程序设计的需求，它对 Visual C#的支持包括项目模板、设计器、属性页、代码向导、一个对象模型以及开发环境的其他功能。Visual C#编程所基于的类库是.NET Framework。

开发工具的获取：

1）使用 .NET Framework SDK。可以到下列地址免费下载：

http://www.microsoft.com/downloads/Search.aspx?displaylang=zh-cn

2）使用 Visual C# 2010/2015 Express Edition。可以到下列地址免费下载

https://msdn.microsoft.com/zh-cn/

1. 系统需求

Microsoft .NET Framework SDK 2.0 版包括了开发人员编写、生成、测试和部署.NET Framework 应用程序所需的工具、文档和示例。

SDK 支持的操作系统包括：Windows 2000 Service Pack 3; Windows Server 2003/2008; Windows Vista; Windows 7; Windows XP Service Pack 2。

2. C#软件环境的安装

1）要运行为.NET Framework 开发的应用程序，必须安装 Microsoft .NET Framework Redistributable Package（可再发行组件包）。.NET Framework 可再发行组件包将安装.NET Framework 运行库，以及运行针对.NET Framework 开发的应用程序所需的相关文件。先将下载的文件保存到计算机上，以后再执行安装程序。文件名为 dotnetfx.exe，文件大小为 22.4 MB。

2）要开发和运行.NET Framework 应用程序，必须安装 Microsoft .NET Framework SDK。先将下载的文件保存到计算机上，以后再执行安装。文件名为 setup.exe，文件大小为 425 MB。

3. 编辑 C#源程序

使用文本编辑器创建 C#程序的源文件，并将其存储为名如 Hello.cs 的文件。C# 源代码文件使用的扩展名是.cs。

以下控制台应用程序是传统“Hello World!”程序的 C# 版，运行该程序将在控制台显示字符串 Hello World!。

```
// A "Hello World!" program in C#
class Hello{
   static void Main(){
      System.Console.WriteLine("Hello World!");
   }
}
```

注意该程序的几个要点：1）代码注释；2）Main 方法；3）输入和输出。这些也是其他编程语言需要具备的基本要素。

4. 编译和运行 C#程序

从命令行编译程序：打开 DOS 窗口，设置好环境变量，进入 C#源程序所在的文件夹（假设为 d:\csharp），然后输入命令：

```
csc Hello.cs
```

如果程序没有包含任何编译错误，则将创建一个名为 Hello.exe 的文件。
若要运行程序，请输入命令：

```
Hello
```

5．命令行编译示例

- 编译 File.cs 以产生 File.exe：

```
csc File.cs
```

- 编译 File.cs 以产生 File.dll：

```
csc /target:library File.cs
```

- 编译 File.cs 并创建 My.exe：

```
csc /out:My.exe File.cs
```

- 编译当前目录中所有的 C# 文件，以产生 File2.dll 的调试版本。不显示任何警告：

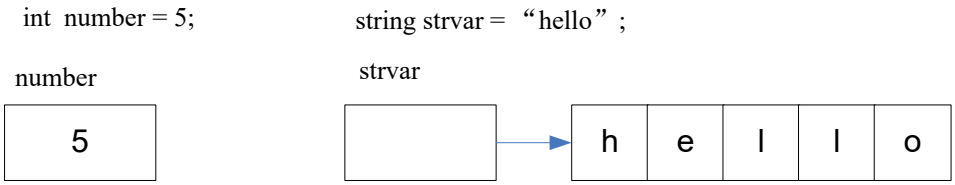
```
csc /target:library /out:File2.dll /warn:0 /debug *.cs
```

有关 C# 编译器及其选项的更多信息，请参见 C# 相关手册中关于编译器选项的说明，也可以在 Visual Studio 的集成环境中通过创建项目来编译“Hello World!”程序，这可以参考 Visual Studio 的使用手册。

1.3.2 C#的数据类型与流程控制

1. 数据类型

C#语言的数据类型主要分为两类：值类型（value type）和引用类型（reference type）。值类型的变量包含其自身的数据，而引用类型的变量包含的是实际数据的引用或者称句柄，即它是对真正包含数据的内存块的指向。从图 1.4 中可以清晰地看出值类型变量和引用类型变量两者的差别。



(a) 值类型变量　　(b) 引用类型变量

图 1.4 值类型变量和引用类型变量

C#语言的值类型包括结构类型和枚举类型（enum）。结构类型包括简单类型（primitive type）和

用户自定义结构类型（struct）。简单类型可分为布尔类型（bool），字符类型（char）和数值类型。表 1.3 是对 C#的简单数据类型的详细描述及示例。

C#语言中 bool 类型严格地与数值类型相互区分，只有 true 和 false 两种取值，它们不能像 C/C++程序里那样与其他类型之间进行转换。C#的 char 类型的字宽为 16 位，表示一个 Unicode 编码。

数值类型包括整数，浮点和 decimal 三种类型。整数类型有 sbyte，byte，short，ushort，int，uint，long，ulong 共八种，它们两两一组分别为有符号和无符号两种，字宽分别为 8 位、16 位、32 位和 64 位。浮点值有 float 和 double 两种不同精度的类型。decimal 主要用于金融、货币等对精度要求比较高的计算环境。

C#语言的引用类型共分四种类型：类（class），接口（interface），数组（array）和委派（delegate）。

利用类（class）程序员可以定义自己的类型，事实上类是自定义类型最重要的渠道。C#中已定义了两个比较特殊的类类型：object 和 string。object 是 C#中所有类型（包括所有的值类型和引用类型）从其继承的根类。string 类型是一个密封类型（不能被继承的类型），该类对象表示用 Unicode 编码的字符串。

接口（interface）类型定义一个有关一系列方法的合同。委派（delegate）类型是一个指向方法的签名，类似于 C/C++中的函数指针。这些类型实际上都是类的某种形式的包装。

表 1.3 C#的简单数据类型关键字及示例

简单类型	描 述	示 例
sbyte	8-bit 有符号整数	sbyte val = 12;
byte	8-bit 无符号整数	byte val1 = 12; byte val2 = 34U;
short	16-bit 有符号整数	short val = 12;
ushort	16-bit 无符号整数	ushort val1 = 12; ushort val2 = 34U;
uint	32-bit 无符号整数	uint val1 = 12; uint val2 = 34U;
int	32-bit 有符号整数	int val = 12;
long	64-bit 有符号整数	long val1 = 12; long val2 = 34L;
ulong	64-bit 无符号整数	ulong val1 = 12; ulong val2 = 34U; ulong val3 = 56L; ulong val4 = 78UL;
float	32-bit 单精度浮点数	float val = 1.23F;
double	64-bit 双精度浮点数	double val1 = 1.23; double val2 = 4.56D;
bool	布尔类型	bool val1 = true; bool val2 = false;
char	字符类型 ，16-bit Unicode 编码	char val = 'h';
decimal	28 个有效数字的 128-bit 十进制类型	decimal val = 1.23M;

每种数据类型都有对应的缺省值。数值类型的缺省值为 0 或 0.0，char 类型的缺省值为'\x0000'，

bool 类型的缺省值为 false。枚举类型的缺省值为 0。结构类型的缺省值是将它所有的值类型的域设置为对应值类型的缺省值，将其所有引用类型的域设置为 null。所有引用类型的缺省值为 null，表示引用类型变量没有对任何实际地址进行引用。

不同类型的数据之间可以转换，C#的类型转换有隐式转换、显式转换、标准转换、自定义转换共四种方式。

隐式转换与显式转换和 C++里一样，隐式转换是系统默认的、不需要任何声明就可以进行的转换，它是由编译器根据不同类型数据间转换规则（“小类型”到“大类型”转换）自动完成的，又称为自动转换。显式类型转换就是强制执行从一种数据类型到另一种数据类型的转换，因此也称为强制类型转换，从“大类型”到“小类型”的转换必须用显式转换，显式转换需要括号转换操作符，也就是使用“（Type）data”形式。

标准转换和自定义转换是针对系统内建转换和用户定义的转换而言的，两者都是对类或结构这样的自定义类型而言的。

值类型和引用类型之间的转换依靠装箱（boxing）和拆箱（unboxing）机制完成。

装箱就是将值类型变量隐式地转换为引用类型变量。把一个值类型装箱的具体过程是：创建一个 object 对象类型的实例，并把该值类型的值复制给该实例。例如：

```
int val = 1000;
object obj = val;                    //装箱，将val的值复制给obj
```

拆箱是装箱的逆过程。具体过程是：先确认 object 类型实例的值是否为目标值类型的装箱值，如果是，则将这个实例的值复制给目标值类型的变量。例如：

```
int val = 1000;
object obj = val;                    //装箱
int val1 = (int)obj;                 //拆箱
```

当进行装箱操作时，不需要显式地进行强制类型转换；而要进行拆箱操作时，必须显式地进行类型转换。

2. 操作符与表达式

C#从 C/C++中继承了几乎所有的操作符，它们主要分为 4 类：算术操作符，位操作符，关系操作符和逻辑操作符。表 1.4 按优先级从高到低的次序列出 C#定义的所有操作符，分隔符的优先级最高，表中“左⇨右”表示从左向右的操作次序（结合性）。

表 1.4 操作符的优先级

优先级	运 算 符	结合性
1	. [] () new typeof sizeof checked unchecked	
2	++ -- ~ ! + －（一元）	右⇨左
3	* / %	左⇨右
4	+ －（二元）	左⇨右
5	<< >>	左⇨右
6	< > <= >= is as	左⇨右
7	== !=	左⇨右
8	&	左⇨右
9	∧	左⇨右
10	|	左⇨右
11	&&	左⇨右
12	||	左⇨右
13	？：	右⇨左

优先级	运　算　符	结合性
14	=　*=　/=　%=　+=　—=　<<=　>>=　&=　∧=　|=	右⇨左

C#引入了几个具有特殊意义的操作符：as，is，new，typeof。

作为操作符的 new 关键字用于创建对象和调用构造方法，值得注意的是，值类型对象（例如结构）是在栈上创建的，而引用类型对象（例如类）是在堆上创建的。new 也用于修饰符，用于隐藏基类的继承成员。为隐藏继承的成员，使用相同名称在派生类中声明该成员并用 new 修饰符修改它。

as 操作符用于执行兼容类型之间的转换，当转换失败时，as 操作符结果为 null。is 操作符用于检查对象的运行时类型是否与给定类型兼容，当表达式非 null 且可以转化为指定类型时，is 操作符结果为 true，否则为 false。as 和 is 操作符是基于同样的类型鉴别和转换方式而设计的，两者有相似的应用场合。实际上 expression as type 相当于 expression is type ? (type)expression : (type)null。

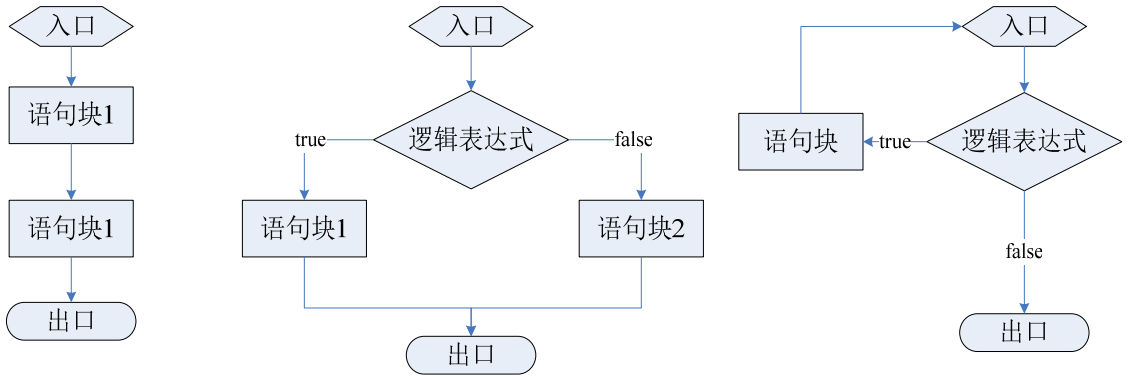
typeof 运算符用于获得某一类型的 System.Type 对象。

C#的某些操作符可以像 C++那样被重载。操作符重载使得自定义类型（类或结构）可以用简单的操作符来方便地表达某些常用的操作。

为完成一个计算结果的一系列操作符和操作数的组合称为表达式。和 C++一样，C#的表达式可以分为赋值表达式和逻辑表达式两种。

3. 流程控制

按程序的执行流程，程序的控制结构可分为 3 种：顺序结构，分支结构和循环结构。这些结构的流程图如图 1.5 所示。



(a) 顺序结构　　(b) if-else分支结构　　(c) while循环结构

图 1.5 三种典型控制结构

1）分支结构语句：

C#有 2 种分支语句实现分支结构，if 语句实现二路分支，switch 语句实现多路分支。这些与 C/C++/Java 别无二致。

if 语句的定义格式如下：

```
if(<布尔表达式>){
    <语句块1>;
}
else{
```

```
        <语句块2>;
    }
```

switch 语句的定义格式如下：

```
    switch(<表达式>) {
        case<常量1>：
            <语句块1>;
            break;
        case<常量2>：
            <语句块2>;
            break;
        ……
        [default: <语句块>;]
    }
```

2）循环语句：

循环语句除了 C/C++/Java 中都具有的 while, do-while, for 三种循环结构外还引入了 foreach 语句用于遍历集合中所有的元素。

for 语句的定义格式如下：

```
    for(int i=0; i<10; i++){
        <语句块>;
    }
```

foreach 语句的定义格式如下：

```
    foreach(string s in args){
        <语句块>;
    }
```

while 语句的定义格式如下：

```
    int i=0;
    while(i<10){
        <语句块>;
        i++;
    }
```

3）转向语句:

跳转语句有 break，continue，goto，return，throw 五种语句，前四种与 C++里的语义相同，throw 语句与后面将介绍的 try 语句一起用来进行异常处理。

4. 数组

C#中数组的工作方式与大多数其他流行语言中数组的工作方式类似，但还是有一些差异应引起注意。C#支持一维数组、多维数组（矩形数组）和数组的数组（又称为交错的数组）。

1）一维数组

声明一维数组变量的格式是:

```
    <类型>[ ] 数组名。
```

例如：

```
    int[] a;
```

注意声明数组时，方括号“[ ]”必须跟在类型后面，而不是标识符后面。另一细节是，数组的大

小不是其类型的一部分，而在 C 语言中它却是数组类型的一部分。因此在 C#中可以声明一个数组并向它分配相应类型对象的任意数组，而不管数组长度如何。

声明数组并没有实际创建它们。在 C#中，数组是对象，必须进行实例化。只有用 new 操作符为数组分配空间后，数组才真正占有实在的存储单元。使用 new 创建一维数组的格式如下：

```
<数组名> = new <类型>[<长度>]
```

可以在声明数组的同时为数组进行初始化，例如：

```
int[] a = {1,2,3,4,5};
```

通过下标可以访问数组中的任何元素。数组元素的访问格式为：

```
<数组名>[<下标>]。
```

C#中的数组都是继承自抽象基类型 System.Array，因此数组实际上是对象。类 System.Array 中具有的属性以及其他类成员都可以供任意数组对象使用，例如 Length 属性可以获得数组元素的个数，即数组的长度。System.Array 类还提供了许多用于排序、搜索和复制数组的方法。

2）多维数组

多维数组用说明多个下标的形式来定义，例如：

```
int[,] items = new int[5,4];
```

声明了一个二维数组 items，并分配 5×4 个存储单元。还可以有更大的数组。例如，可以有三维的矩形数组：

```
int[,,] buttons = new int[4,5,3];
```

同样也可以初始化多维数组，例如：

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
```

可以省略数组的大小，如下所示：

```
int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
```

3）数组的数组

如果数组的元素也是数组，则称为数组的数组，或交错的数组。例如：

```
int[][]  scores = new int[4][];
```

声明并分配交错数组 scores，它具有四个元素 scores[0], …, scores[3]，每个元素都是一个整型数组，即 int[ ]，每个数组的长度可以不一样。

【例1.6】 写一个完整的 C#程序，声明并实例化三种数组。

```
using System;
class ArraysSample {
    public static void Main() {
        int[] numbers = new int[5];                  // 一维数组
        string[,] names = new string[5, 4];          // 多维数组
        Console.WriteLine(names.Length);
        int[][] scores = new int[4][];               // 数组的数组（交错数组）
        for (int i = 0; i < scores.Length; i++) {
            scores[i] = new int [i + 3];             // 创建交错数组
        }
        for (int i = 0; i < scores.Length; i++) {
            Console.WriteLine("Length of row {0} is {1}", i, scores[i].Length);
        }
        Console.WriteLine("Number of rows is {0}",scores.Length);
```

```
        }
    }
```

程序输出结果如下：

```
20
Length of row 0 is 3
Length of row 1 is 4
Length of row 2 is 5
Length of row 3 is 6
Number of rows is 4
```

1.3.3 类与对象

C#在面向对象技术的基础上引入一些新的特性，使其特别适合组件编程。组件编程技术不是对传统面向对象技术的抛弃，相反组件编程正是面向对象编程的深化和发展。类作为面向对象的灵魂在 C#语言里有着相当广泛深入的应用。

1. 类的声明

C#的类（class）是一种封装包括数据成员，方法成员和嵌套类型等多种成员的数据结构。其中数据成员可以是常量和域（又称字段成员，field member）；方法成员（method member）可以是方法（method），属性（property），索引器（indexer），事件（event），操作符方法（operator），构造方法（construtor）。

在 C#语言中，类的定义格式包括两个部分：类声明和类主体。类声明包括关键字 class、类名及类的属性。类声明的格式如下：

```
[<修饰符>] class <类名> [: <基类名>]
```

包含类主体的类结构如下：

```
<类声明>{
    <数据成员声明>
    <方法成员声明>
}
```

声明常量成员要用 const 关键字，并给出常量名及其类型。其格式如下：

```
[<修饰符>] const <常量类型> <常量名> = <值>
```

声明字段成员必须给出变量名及其所属的类型，同时还可以指定其他特性。其格式如下：

```
[<修饰符>] [static]  <变量类型> <变量名>
```

声明方法成员的格式如下：

```
[<修饰符>] [static]  <返回值类型> <方法名> (<参数列表>) {
    <方法体>
}
```

构造方法是一种特殊的方法成员，它具有与类名相同的名字，用于对类的实例进行初始化，构造方法声明中无需返回值类型。

属性提供对类的某个特征的访问。一个属性可以有 2 个访问操作符，分别是 get 和 set，它们分别指定属性读取或写入新值的方式。属性定义的格式如下：

```
[<修饰符>] [static]  <类型> <属性名>{
    get {属性读取过程}
    set {写入新属性值的过程}
}
```

属性的声明方式和方法非常相似，区别在于它没有使用括号，也不能使用显示的参数。

在类的成员声明中，用 static 关键字声明静态成员，静态成员属于整个类，该类的所有实例共享静态成员。实例成员的声明没有 static 关键字，该类的每个实例都有自己专有的实例成员。

下面的代码给出复数 Complex 类的定义。C#语言与 C 和 C++等语言一样，都没有将复数设计为内部数据类型，而在科学与工程数值计算中，复数运算是基本运算之一。当我们需要频繁操作复数时，需要自定义复数类。

```
public class Complex: IComparable {
    private double rp = 0.0;              // 复数的实部
    private double ip = 0.0;              // 复数的虚部
    private static double eps = 0.0;      // 缺省精度
    public double RealPart {
        get { return rp; }
        set {rp = value;}
    }
    public double ImaginaryPart {
        get {return ip;}
        set {ip = value;}
    }
    public Complex() { }
    public Complex(double r, double i) {
        rp = r;
        ip = i;
    }
    ..... // 实现复数操作的其他相关方法，如加减乘除、指数对数、三角函数等。
}
```

2. 对象的创建和使用

对象是类的实例，属于某个已知的类。因此定义对象之前，一定要先声明该对象所属的类。对象声明的格式为：

```
<类名>  <对象名>;
```

对象的声明并没有实际创建新的对象，需要用 new 操作符在需要的时候创建新的对象，并为之

分配内存。new 操作符调用类的构造方法来初始化新的对象。

在声明对象的同时可以使用 new 操作符创建对象，其格式如下：

〈类名〉〈对象名〉= new〈与类名相同的构造方法〉(〈参数列表〉);

例如：

Stack s = new Stack();

通过对象引用类的成员变量的格式为：

〈对象名〉.〈成员名〉

通过对象调用成员方法的格式为：

〈对象名〉.〈方法名〉（〈参数列表〉）

不同于 C++，在 C#程序中，程序员只需创建所需对象，而不需显式地销毁它们。.NET 平台的垃圾回收（gabage collect，GC）机制自动判断对象是否还在被使用，并能够自动销毁不再使用的对象，收回这些对象所占用的内存资源。

3. 类成员的访问权限控制

C#程序用多种访问修饰符来表达类中成员的不同访问权限，以实现面向对象技术所要求的抽象、信息隐藏和封装等特性。

信息隐藏和封装特性要求将类设计成一个黑匣子，只有类中定义的公共接口对外部是可见的，而类实现的细节一般是不可见的，类的使用者不能直接对类中的数据进行操作，这样可以防止外界的误用或干扰。即使类的设计者要改变类中数据的定义，只要外部接口保持不变，就不会对使用该类的程序产生任何影响。因此信息隐藏和封装减少了程序对类中数据表达的依赖性。

C#为类的成员定义了五种不同的访问权限限制修饰符，如表 1.5 所示，缺省访问权限为 private。

命名空间或组合体（程序集）内的类有 public 和 internal 两种修饰，缺省方式为 internal，本章后面有对命名空间和组合体的简单解释。

表 1.5 权限修饰符允许的访问级别

权限修饰符	本类	子类	本组合体	其他类
公有的（public）	✓	✓	✓	✓
保护的（protected）	✓	✓		
内部的（internal）	✓	✓	✓	
私有的（private）	✓			

4. 实例成员和类成员

C#类包括两种不同类型的成员：实例成员和类成员，类成员也称为静态成员。C#所有的对象都将创建在托管堆上。当创建类的一个对象时，每个对象拥有了一份自己特有的数据成员拷贝。这些为特有的对象所持有的数据成员称之为实例数据成员，没有用关键字 static 修饰的成员就是实例成员。相反那些不为特有的对象所持有的数据成员称为类成员，或称为静态数据成员，在类中用

static 修饰符声明。不为特有的对象所持有的方法成员称为静态方法成员。C#中静态数据成员和静态方法成员只能通过类名引用获取。

1.3.4　类的继承与嵌套

继承（inheritance）是面向对象技术的关键特性之一，面向对象编程语言都提供类的继承机制。从现有类出发定义一个新类，我们称新类继承了现有的类。被继承的类叫做基类（base class）或父类，继承的类叫做基类的派生类（derived class）或子类。

C#程序中所有的类都直接或间接继承自.NET Framework 的 System.Object 类，这个 System 命名空间中的 Object 类完全等同于用小写的 object 关键字来表示的类。我们定义一个类时，如果没有明确指定它的基类，编译器缺省认为该类继承自 object 类。这样 C#中所有的类都继承了 System.Object 类的公共接口，剖析它们对我们理解并掌握 C#中类的行为非常重要。System.Object 类的公共接口如下所示：

```
namespace System{
    public class Object {
        public static bool Equals(object objA, object objB) { }
        public static bool ReferenceEquals(object objA, object objB) { }
        public Object() { }
        public virtual bool Equals(object obj) { }
        public virtual int GetHashCode() { }
        public Type GetType() { }
        public virtual string ToString() { }
        protected virtual void Finalize() { }
        protected object MemberwiseClone() { }
    }
}
```

1. 派生类的声明

在类的声明时可以说明类的基类，声明格式如下：

```
public class <派生类名>:<基类名>
```

在C#程序中，除object类之外的每个类都有基类，如果没有显式地标明基类，则隐式地继承自 object 类。

派生类继承基类中所有可被派生类访问的成员，当派生类成员与基类成员同名时，称派生类成员将基类同名成员隐藏，这时应在派生类成员定义中加上 new 关键字。

对基类中声明的虚方法（virtual method），派生类可以重写（override），即为声明的方法提供新的实现。这些方法在基类中用 virtual 修饰符声明，而在派生类中，将被重写的方法用 override 修饰符声明。

2. this 与 base 引用

在 C#程序中，可以通过 this 引用访问每个对象自身的成员，通过 base 引用访问从基类继承的成员。

3. 嵌套类

一个类或其他类型可以嵌套定义于另一个类中，称为嵌套类或嵌套类型。与一般的类相同，嵌套类可以具有成员变量和成员方法。通过建立嵌套类的对象，可以存取其成员变量和调用其成员方法。

1.3.5 抽象类、密封类和接口

1. 抽象方法与抽象类

声明类时，除了可以说明类的基类，还可以声明抽象类（abstract class）或密封类（sealed class）等特性。当需要定义一个抽象概念时，可以声明一个抽象类，该类只描述抽象概念的结构，而不实现每个方法。这个抽象类可以作为一个基类被它的所有派生类共享，而其中的方法由每个派生类去实现。

当声明一个方法为抽象方法（abstract method）时，则不需提供该方法的实现，但这个方法或者被派生类继续声明为抽象的，或者被派生类实现。用关键字 abstract 来说明抽象方法，例如：

```
abstract void f1();
```

任何包含抽象方法的类必须被声明为抽象类，抽象类是不能直接被实例化的类。用关键字 abstract 来声明抽象类，例如：

```
abstract class AC1{...}
abstract class AC2: AC1{...}
```

抽象类的派生类必须实现基类中的抽象方法，或者将自己也声明为抽象的。

2. 密封类

密封类是指不能被继承的类，即密封类不能有派生类。用关键字 sealed 来说明密封类，例如：

```
sealed class SC{...}
```

3. 接口

接口（interface）类似于类（class），但它仅说明方法的形式，不需定义方法的实现，可以说接口是定义一组方法的合同。接口的定义形式如下：

```
[<修饰符>] interface <接口名> [:<基接口名>] {
    <方法1>
    <方法2>
}
```

接口的所有成员都定义为公共（public）成员，并且接口不能包含常量、字段（私有数据成员）、构造方法及任何类型的静态成员。一旦定义了一个接口，一个或更多的类就能实现（implement）这

个接口，即这个类将满足接口规定的合同。实现某接口的类声明的格式如下：

```
[<修饰符>] class <类名> : [<基类名>], [<接口名>] {...}
```

C#只支持单重继承机制，即一个类只能继承于一个基类，但是一个类可以实现多个接口。

C#类库中已定义了 IComparable 接口，它的定义如下：

```
public interface IComparable {
//  将当前实例与同一类型的另一个对象进行比较，并返回一个整数，该整数指示当前实例在排序顺序中的位置是位于另一个对象之前、之后还是与其位置相同。
// 返回结果: 一个值，指示要比较的对象的相对顺序。返回值的含义如下：返回值小于零此实例小于obj；返回值等于零，此实例等于obj。返回值大于零，此实例大于 obj。
    int CompareTo(object obj);
}
```

可进行比较操作的类型都应实现该接口，即在类的设计中，完成方法 CompareTo 的具体定义。几乎所有内部数据类型也都实现了该接口，因而成为可比较的类型，如 int，double，string 等等。

4. 多态性

在面向对象编程语言中，多态性（polymorphism）是指“一个接口，多个方法”，即一个方法可能有多个版本，一次单独的方法调用（invoke）可能是这些版本中的任何一种。多态性有两种表现形式：方法的重载和方法的覆盖

1) 方法的重载（method overloading）

一个类中如果有许多同名的方法带有不同的参数，称为方法的重载。例如，控制台输出方法 Console.Write 可以有不同的参数：

```
Console.Write(string s);
Console.Write(int a);
```

一个方法的名称和它的形参的个数和每个形参的类型组成该方法的签名（signature），可以有多个同名的方法，但每个方法的签名应该是唯一的，所以在方法重载时要注意以下两点：

- 参数必须不同：可以是参数个数不同，也可以是参数类型不同，或者是参数顺序不同。
- 方法的签名不取决于方法的返回类型，即仅利用不同的返回类型无法区分方法。

方法重载的价值在于，它允许通过使用一个普通的方法来访问一系列相关的方法。当调用一个方法时，具体调用哪一个版本则根据调用方法的参数由编译程序决定，编译程序将选择与调用的实参相匹配的重载方法。

2) 方法的覆盖

C#通过为方法的定义引入 virtual（虚方法）和 override（方法覆盖/重写）关键字提供父子类间方法继承的机制。类的虚方法是可以在该类的继承类中改变其实现的方法，当然这种改变仅限于方法体的改变，而非方法头（方法声明）的改变。被子类改变的虚方法必须在方法头加上 override 关键字来表示。当一个虚方法被调用时，该类的实例的运行时类型(run-time type)决定哪个方法体被调用。

进行方法覆盖，子类必须覆盖父类中声明为 virtual 的方法；子类覆盖父类中的虚方法时，子类方法必须与父类中的方法有相同的方法签名。

值得指出的是虚方法不可以是静态方法 — 也就是说不可以用 static 和 virtual 同时修饰一个方法，这由它的运行时类型辨析机制所决定。override 必须和 virtual 配合使用，当然也不能和 static 同时使用。

如果在一个类的继承体系中不想再使一个虚方法被覆盖，我们可以将 sealed 和 override 同时修饰一个虚方法。例如：sealed override public void F()。注意这里一定是 sealed 和 override 同时使用，也一定是密封覆盖一个虚方法，或者一个被覆盖（而不是密封覆盖）了的虚方法。

1.3.6 命名空间和程序集

除了依赖于几个系统提供的类(如 System.Console)，到目前为止介绍的程序都是独立存在的。但更常见的情况是：实际的应用程序由若干不同的部分组成，每个部分分别进行编译。例如，企业级应用程序可能依赖于若干不同的组件，其中包括某些内部开发的组件和某些从独立软件供应商处购买的组件。

命名空间（name space）和程序集（assembly）有助于开发基于组件的系统。每个命名空间定义一个声明空间，用于声明多个类型（类、结构、接口、枚举等）或嵌套的命名空间。命名空间为应用程序的组织提供一种逻辑体系，它既用作程序的“内部”组织体系，也用作“外部”组织体系，即它提供了一种向其他程序表示公开程序元素的途径。

程序集，或称作组合体，用于物理打包和部署程序。程序集可以包含类型、用于实现这些类型的可执行代码以及对其他程序集的引用。

有两种主要的程序集：应用程序和库。应用程序有一个主入口点，通常具有 .exe 文件扩展名；而库没有主入口点，通常具有 .dll 文件扩展名。

为了说明命名空间和程序集的使用，我们再次以“hello world”程序为例，并将它分为两个部分：提供消息的类库和显示消息的控制台应用程序。

这个类库仅含一个名为 HelloMessage 的类。

```
// HelloLibrary.cs
namespace CSharp.Introduction {
      public class HelloMessage {
          public string Message {
              get { return "hello world"; }
          }
      }
 }
```

该类定义在名为 CSharp.Introduction 的命名空间中。HelloMessage 类提供一个名为 Message 的只读属性。命名空间可以嵌套，而声明

```
namespace CSharp.Introduction {...}
```

仅是若干层命名空间嵌套的简写形式。若不简化，则应该像下面这样声明：

```
namespace CSharp {
    namespace Introduction
    {...}
}
```

将“hello world”组件化的下一个步骤是编写使用 HelloMessage 类的控制台应用程序。可以使用此类的完全限定名 CSharp.Introduction.HelloMessage，但该名称太长，使用起来不方便。一种更方便的方法是使用“using 命名空间”指令，这样，使用相应的命名空间中的所有类型时就不必加限定名称。例如：

```
using CSharp.Introduction;
class HelloApp {
    static void Main() {
        HelloMessage m = new HelloMessage();
        System.Console.WriteLine(m.Message);
    }
}
```

该例通过“using 命名空间”指令引用 CSharp.Introduction 命名空间。这样，HelloMessage 就成为 CSharp.Introduction.HelloMessage 的简写形式。

C#还允许定义和使用别名。“using 别名”指令定义类型的别名。当两个类库之间发生名称冲突时，或者当使用大得多的命名空间中的少数类型时，这类别名很有用。示例

```
using MessageSource = CSharp.Introduction.HelloMessage;
```

将 MessageSource 定义为 HelloMessage 类的别名。

我们已编写的代码可以编译为包含类 HelloMessage 的类库和包含类 HelloApp 的应用程序。使用 Visual Studio 2008/2010 中提供的命令行编译器 csc 时，用如下所列的命令：

```
csc /target:library HelloLibrary.cs
csc /reference:HelloLibrary.dll HelloApp.cs
```

第一条命令产生一个名为 HelloLibrary.dll 的类库，第二条命令产生一个名为 HelloApp.exe 的应用程序。

当在 Visual Studio 中构建项目时，我们实际上就是旨在创建.NET 程序集。从形式上说，程序集就是一个物理文件（其文件扩展名通常是 *.exe 或 *.dll），我们可以使用 Windows Explorer 在硬盘驱动器上直接查看此文件。

Visual Studio 解决方案浏览器窗口将显示一个名为 References 的子文件夹，它列出了当前项目所使用的程序集的集合。不同的项目需要引用的程序集可能是不一样的，图 1.6 显示了当前控制台应用程序的程序集。

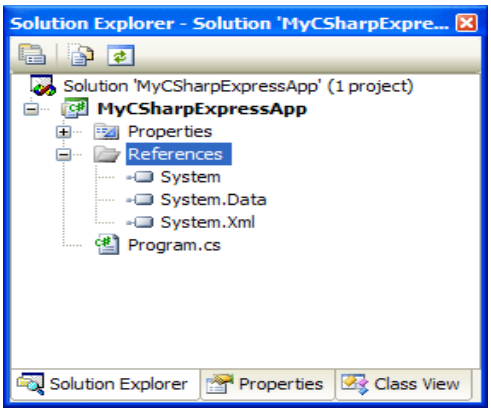


图 1.6 控制台应用程序项目引用的程序集

随着构建越来越大的.NET 应用程序，程序员通常需要使用特定项目所包含的集合以外的程序集。为此，Visual Studio 提供了“添加引用”对话框，它可以使用 Project | Add Reference 菜单命令调用。

程序集可以包含一组命名空间。简单地说，命名空间就是一个在语义上相关的类型的集合。单个程序集通常会包含多个命名空间。例如，mscorlib.dll 程序集就包含文件输入和输出(System.IO)、集合类型 (System.Collections)、通用实用工具类型 (System) 等等命名空间。一个命名空间内可以定义任意数量的类型。

当某个 C#项目需要使用给定程序集中的类型时，第一步就是使用“Add Reference”对话框来引用相应的程序集（*.dll）。第二步是在 C#源文件的开头添加 using 指令，以指定要访问的命名空间。这样我们就能在代码中只使用 List 来代替全名 System.Collections.Generic.List。

在后面的章节中我们将设计和实现多种常见的数据结构和算法。管理这些程序的最简单方法，是将有关数据结构定义在独立的类模块中，所有模块集中在一个文件夹中，不指定命名空间，也就是使用缺省的命名空间。随着编写的代码越来越多，有必要采用更为系统的管理代码的方法。为了使读者在某一章的学习中将注意力集中在当前章节，我们在 Visual Studio 中为每一章设立两个项目：一个项目为“类库”型项目，用于实现相关数据结构的基础类定义，这些类都声明在命名空间 DSAGL 中；一个项目为“控制台应用程序”型项目，用于实现相关数据结构的测试、演示和应用，一般需要引用相应的类库模块，并在测试和应用代码中加入“using DSAGL”指令，以方便源代码的编辑。例如在第二章中，用名为 lists 的项目实现各种“线性表”数据结构，用名为 liststest 的项目实现多个使用“线性表”数据结构的应用程序。在熟悉了相关概念后，用.NET Framework SDK 的命令行编译器也能顺利完成各章代码的编译和运行。

1.3.7 异常处理

在程序执行的过程中，无论什么时候出现了严重错误，.NET 运行环境都会创建一个 Exception 对象来描述和处理该错误。在.NET 平台中，Exception 是所有异常类的基类。从 Exception 基类派生了两种类别的异常：System.SystemException 和 System.ApplicationException。System 命名空间中的所有异常类型都是从 SystemException 派生的，而用户定义的异常应该从 ApplicationException 派生，以便区分运行库错误和应用程序错误。一些常见的系统异常包括：

- IndexOutOfRangeException — 使用了大于数组或集合大小的索引
- NullReferenceException — 在将引用设置为有效的实例之前使用了引用的属性或方法
- ArithmeticException — 在操作产生溢出或下溢时引发的异常
- FormatException — 参数或操作数的格式不正确

与 Java 编程语言中一样，当我们有一段容易引起异常的代码时，我们应该将此代码放在 try 语句块中，紧接其后的是一个或多个提供错误处理的 catch 语句块。如果 try 块中的某一操作触发了一个异常，则错误将传递到相关的 catch 作用域，可以在该作用域中适当地处理此问题。如果 try 作用域中的每个语句在执行时都没有出现错误，那么将跳过整个 catch 块。我们还可以对任何我们想执行但又不知道是否引发异常的代码使用 finally 块。

当使用多个 catch 块时，捕获异常的代码必须以升序的顺序放置，这样就只有第一个与引发的异常相匹配的catch块会被执行。C# 编译器会强制这样做，而 Java 编译器不做这种强制检查。C# 也与 Java 一样，catch 块可以不需要参数；在缺少参数的情况下，catch 块适用于任何 Exception 类。

例如，当从文件中进行读取时，可能会遇到 FileNotFoundException 或 IOException，我们需要首先放置更具体的 FileNotFoundException 处理程序，然后放置更一般的 IOException 处理程序。

```
try {
    string myText = File.ReadAll(@"D:\myTextFile.txt");
}
catch (FileNotFoundException ex){
     Console.WriteLine("Error: {0}", ex.Message);
}
catch (IOException ex){
     Console.WriteLine("Error: {0}", ex.Message);
}
```

1.3.8　C#的标准输入流和输出流

System.Console 类对从控制台读取字符并向控制台写入字符的应用程序提供基本支持。来自控制台的数据从标准输入流读取；传给控制台的正常数据会写入标准输出流；而传给控制台的错误数据会写入标准错误输出流。应用程序启动时，这些流自动与控制台（Console）关联，并以 In、Out 和 Error 属性形式提供给程序员。

默认情况下，In 属性的值是 System.IO.TextReader 对象，而 Out 和 Error 属性的值是 System.IO.TextWriter 对象。

1. Console 类包含的控制台输入方法

从标准输入流读取下一个字符：public static int Read();

从标准输入流读取下一行字符：public static string ReadLine();

2. Console 类包含的控制台输出方法

Console 类包含若干“Write”方法，这些方法可自动将各种值类型的实例、字符数组或对象转换为格式化字符串或无格式字符串，然后将该字符串写入控制台，该字符串后面还可以带行终止字符串。

public static void WriteLine(char);	public static void Write (char);
public static void WriteLine(int);	public static void Write (int);
public static void WriteLine(double);	public static void Write (double)
public static void WriteLine(string);	public static void Write (string);

使用指定的格式信息，将指定的对象数组（后跟当前行结束符）写入标准输出流：

public static void WriteLine(string format, params object[] args);

还可以显式调用 In、Out 和 Error 三种属性所表示的流对象的方法和属性成员。例如，Console.Error.WriteLine 方法将字符串输出到标准错误输出流。特别是，Console 中没有任何方便方法可以将数据写入标准错误输出流，但可以使用 Console.Error.WriteLine 的显式调用实现这一任务。

System.IO 命名空间包含允许在数据流和文件上进行读取及写入的类型。文件和流的差异在于，文件是一些具有永久存储及特定顺序的字节组成的一个有序的、具有名称的集合。对于文件，我们常会想到目录路径、磁盘存储、文件和目录名等方面。相反，流提供一种向某种存储媒介写入字节和读取字节的方式。正如除磁盘外存在多种存储媒介一样，除文件流之外也存在多种流。例如，还存在网络流、内存流和磁带流等。

抽象基类 Stream 支持读取和写入字节。所有表示流的类都是从 Stream 类继承的。Stream 类及其派生类提供数据源和储存库的一般性功能的定义，使程序员不必了解操作系统和基础设备的具体细节。

流涉及三个基本操作：

- 可以从流读取。读取是从流到数据结构（如字节数组）的数据传输。
- 可以向流写入。写入是从数据源到流的数据传输。
- 流可以支持查找。查找是对流内的当前位置进行查询和修改。

1.3.9 C# 2.0 中的泛型

2.0 版 C#语言和公共语言运行库 (CLR) 中增加了泛型（Generics），包括泛型类和泛型方法。泛型类和泛型方法同时具备可重用性、类型安全和效率等方面的优点，这是非泛型类和非泛型方法无法具备的。

泛型将类型参数的概念引入.NET Framework，类型参数使得设计如下类和方法成为可能：这些类和方法将一个或多个类型的指定推迟到声明并实例化该类或调用方法的时候。

泛型通常与集合以及作用于集合的方法一起使用。.NET Framework 2.0 版类库提供一个新的命名空间 System.Collections.Generic，其中包含几个新的基于泛型的集合类，如 List<T>。建议面向 2.0 版的所有应用程序都使用新的泛型集合类，而不要使用旧的非泛型集合类，如 ArrayList。有关更多信息，请参见 .NET Framework 类库中的泛型编程指南。

例如，声明并构造整型数的列表：

```
List<int>  a = new List<int>(); // 声明并构造整型数的列表
a.Add(86); a.Add(100);          // 向列表中添加整型元素
```

声明并构造字符串列表：

```
List<string>  s = new List<string>();     // 声明并构造字符串列表
s.Add("Hello"); s.Add("C# 2.0");          // 向列表中添加字符串型元素
```

也可以声明并构造自定义类型的列表：

```
List< Student>  st = new List< Student>(); // 声明并构造学生列表
st.Add(new Student ("200518001", "王兵")); // 向列表中添加学生类型元素
```

当然，也可以创建自定义泛型类型和方法，以提供自己的通用解决方案，设计类型安全的高效模式。在后面的章节中我们所实现的数据结构和算法都是基于泛型的。

C#语言中泛型的优越性在下面的一段例子中应能较好的显示出来。对于同样的运算逻辑（例子中是交换两个变量的内容），但仅是数据的类型不一样，可能就需要定义一堆相似的方法；而应用泛型特性则可仅定义一个泛型方法（例子中是 swap<T>）。

```
static void Main(string[] args) {
     int a = 3;  int b = 7;

    swapint(ref a, ref b);
    double ad = 3.5; double bd = 7.5;
    swapdouble(ref ad, ref bd);

    swap<int>(ref a, ref b);
}

static void swapint(ref int a, ref int b) {
    int x = a;
    a = b;
    b = x;
}

static void swapdouble(ref double a, ref double b) {
    double x = a;
    a = b;
    b = x;
}
```

```
static void swap<T>(ref T a, ref T b) {
    T x = a;
    a = b;
    b = x;
}
```

习题 1

1.1 数据结构课程研究的内容是什么？其中哪个方面独立于计算机？

1.2 数据结构按逻辑结构可分为哪几类？分别有什么特征？

1.3 为什么要进行算法分析？算法分析主要研究哪几个方面？

1.4 分析下面各程序段的时间复杂度。

```
(1)  for(i=0; i<n; i++)
         for (j=0; j<m; j++)
             A[i,j] = 0;
(2)  s=0;
     for(i=0; i<n; i++)
         for(j=0; j<n; j++)
             s += B[i,j];
     sum = s;
(3)  x = 0;
     for(i=1; i<n; i++)
         for (j=1; j<=n-i; j++)
             x++;
(4)  i = 1;
     while(i<=n)
         i=i*3;
```

1.5 数据结构被形式地定义为（D, R），其中 D 是数据元素的有限集合，R 是 D 上的关系的有限集合。设有数据逻辑结构 S=（D, R），试按各小题所给条件画出它们的逻辑结构图，并确定相对于关系 R，哪些结点是起始结点，哪些结点是终端结点？

(1) D = {d1,d2,d3,d4}，R = {(d1, d2), (d2, d3), (d3, d4) }

(2) D = {d1, d2, …, d9}，R = {(d1,d2), (d1,d3), (d3,d4), (d3,d6), (d6,d8), (d4,d5), (d6,d7), (d8,d9) }

(3) D = {d1, d2, …, d9}，R={(d1,d3), (d1,d8), (d2,d3), (d2,d4), (d2,d5), (d3,d9), (d5,d6), (d8,d9), (d9,d7), (d4,d7), (d4,d6)}