目　录

第1章 概述

1.1 计算机的工作模型

自 1946 年第一台电子计算机(ENIAC)问世以来，计算机在理论、技术以及应用等方面有了很大的发展，特别是计算机的应用，它已从早期的数值计算拓广到现在的大量的非数值计算，如：管理信息系统、文字处理系统等都属于计算机在非数值计算方面的应用。现在，计算机已经渗透到人类社会活动的各个领域并发挥着巨大的作用。

一台计算机由硬件和软件二部分构成。硬件是指计算机的物理构成，即构成计算机的元器件和设备。软件是指计算机程序以及相关的文档资料。硬件是计算机的物质基础，软件是计算机的灵魂。没有硬件就没有计算机，但是，如果只有硬件没有软件，可以说计算机什么事情也做不了。要想用计算机来解决各种问题，必须要有相应的软件。从某种意义上讲，一台计算机的性能主要由硬件决定，而它的功能主要是由软件来提供的。

计算机的应用领域在不断扩展，应用的规模、层次和类型也在不断扩大，社会对计算机软件的需求急剧增长。如何设计出大量的满足用户需求的高质量软件是软件工作者所面临的严峻挑战。计算机程序不同于其它程序（如：音乐会程序），它是由计算机来执行的，计算机程序的编制（程序设计）通常要按照计算机解决问题的方式来进行。因此，要进行程序设计，就必须对计算机的工作模型有一定的了解。

1.1.1 硬件结构

虽然计算机有了很大的发展，但目前大部分计算机基本上采用的还是传统的冯•诺依曼（Von Neumann）体系结构，即存储程序式结构。图 1.1 给出了冯•诺依曼计算机的硬件构成。

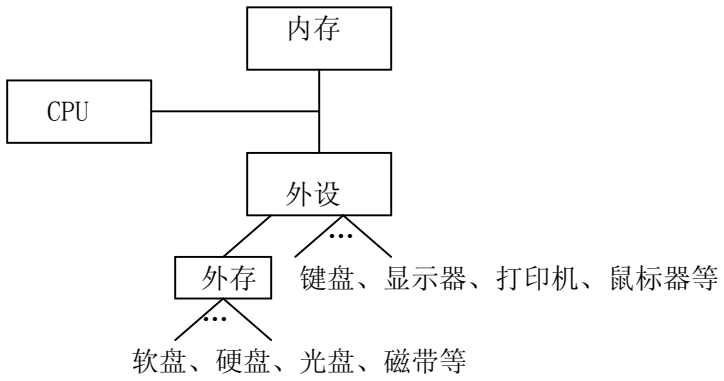


图 1.1 典型的计算机硬件组成

1、中央处理器，简称 CPU（Central Processing Unit）

CPU 是计算机的核心，用于执行计算机指令以实现对整个计算机的控制。它由控制器、运算器以及寄存器等构成。控制器负责取指令和根据指令发出控制信号以引起其它部件的动作。运算器执行运算指令所规定的运算。寄存器用于暂时保存指令的计算结果供下一（几）条指令使用，其作用是减少访问内存的次数，提高指令执行效率。

2、内部存贮器或主存贮器，简称内存（Memory）

内存用于存贮计算机程序（指令和数据）。内存由许多存贮单元构成，存贮单元的大小视计算机而定，一般为一个字节（Byte）。每个存贮单元都有一个地址，对存贮单元的访问是通过其地址来进行的。

3、外部设备，简称外设（Device）

外设用于计算机的输入/输出和提供大容量的信息存贮。它包括输入/输出设备和外部存储器或辅助存贮器（简称外存）。

输入/输出设备提供了计算机与用户的接口，用于用户数据的输入和程序执行结果的输出。键盘和鼠标器等属于输入设备；显示器和打印机等属于输出设备。

外存是大容量的低速存贮部件（与内存相比），用于永久性地存贮程序、数据以及各种文档等信息，包括软盘、硬盘、光盘、磁带等。存贮在外存中的信息通常以文件形式进行组织和访问。内存与外存除了容量和速度不同外，它们的另一个区别在于：内存中存贮的是正在运行的程序和数据。

冯•诺依曼计算机的工作模型是：待执行的程序从外存装入到内存中，CPU 从内存中逐条地取程序中的指令执行。程序执行中所需要的数据从内存或从外设中获得，程序执行中产生的临时数据保存在内存中，程序的执行结果通过外设输出，程序执行结束后从内存退出。

CPU 所能执行的指令通常有：

- 算术指令。实现加、减、乘、除等运算。
- 比较指令。比较两个操作数的大小。
- 数据传输指令。实现 CPU 的寄存器、内存以及外设之间的数据传输。
- 执行流程控制。确定下一条指令的内存地址，包括转移、循环以及子程序调用/返回。通常情况下，CPU 从某个内存地址开始依次取指令来执行。

在冯•诺依曼计算机中存在着几个影响程序执行效率的瓶颈，其中包括 CPU 与内存之间以及内存与外设之间的数据传输。现在的计算机中往往在 CPU 中为内存提供了高速缓存（memory cache）；在内存中为外存提供了高速缓存（disk cache），以解决 CPU 与内存以及内存与外存之间速度不匹配问题。

1.1.2 软件

计算机硬件为计算机提供了物质基础，但它必须通过计算机软件来发挥作用。计算机软件是计算机系统中的程序以及有关的文档。程序是计算任务的处理对象与处理规则的描述；文档是为了便于人理解程序所需的资料说明，供程序开发与维护使用。

软件可以分为：系统软件、支撑软件和应用软件。系统软件居于计算机系统中最靠近硬件的一级，它与具体的应用领域无关，其它软件一般要通过系统软件发挥作用，如操作系统就属于系统软件。支撑软件是指支持软件开发与维护的软件，一般由软件开发人员使用，如软件开发环境就是典型的支撑软件。应用软件是指用于特定领域的专用软件，如：人口普查软件、财务软件等。图 1.2 给出了计算机软件的分类。

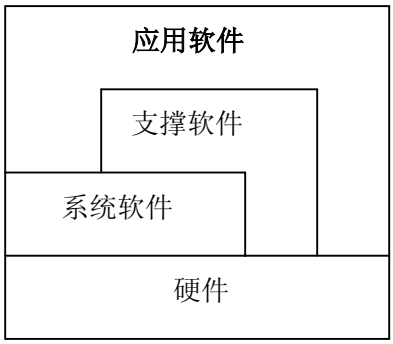


图 1.2 计算机软件的分类

一个软件从无到有，一直到最后的消亡（报废），通常要经历一个过程，这个过程称为软件的生命周期，它包括若干阶段：软件需求分析、软件设计、编程实现、测试以及运行与维护。软机需求分析的主要任务是明确待实现的软件要解决什么问题，即做什么。软件设计是根据软件的需求说明给出抽象的解决方案，它包括概要设计和详细设计。概要设计是指软件的整体结构设计；详细设计是指抽象的数据/算法描述；编程实现是指根据软件设计说明，采用某种程序设计语言书写程序；测试是对书写好的程序进行测试，确认其是否满足所规定的需求；运行与维护是指使用软件并对软件进行维护，其中的维护包括：正确性维护、完善性维护和适应性维护。

早期的软件开发工作主要花费在编程实现阶段，并且采用的是个体的小作坊开发模式。随着计算机应用领域的不断扩大和应用层次的不断加深，使得软件的规模不断扩大、软件的复杂度不断提高，早期的软件开发模式难以驾驭软件开发过程，程序的正确性难以保证，软件生产率急剧下降，出现了“软件危机”。为了解决软件危机，软件工程概念应运而生，其主要思想是采用工程方法来开发软件。在软件工程中，软件开发工作的中心从实现阶段转移到的软件需求分析、设计和维护阶段，并且强调对软件开发过程的管理和加强各个阶段的文档制作。方法和工具构成了软件工程的两大支柱，它们贯穿于软件开发过程，对软件工程思想提供具体的支持。

1.2 程序设计

简单地说，程序设计就是为计算机编制程序的过程，它涉及程序设计方法和程序设计语言等方面的内容。从现代软件工程的角度讲，程序设计是指编程实现阶段的工作，而实际上，我们现在所说的程序设计是指传统意义下的程序设计，它包含软件工程中其

它阶段的一些工作，只是更多地考虑实现技术而已。因此，不能把程序设计仅仅理解成用某种语言来实现设计好的软件，其中还必须考虑需求分析、软件设计、测试以及维护等一些问题。

1.2.1 程序设计风范（Programming Paradigm）

以不同的方式来给出计算的描述就形成了不同的程序设计风范（Programming Paradigms）。具体地讲，程序包括数据以及对数据的加工两部分，程序设计风范就是指以何种观点和方式来看待、组织和描述它们。不同的程序设计风范支持不同的程序设计方法。

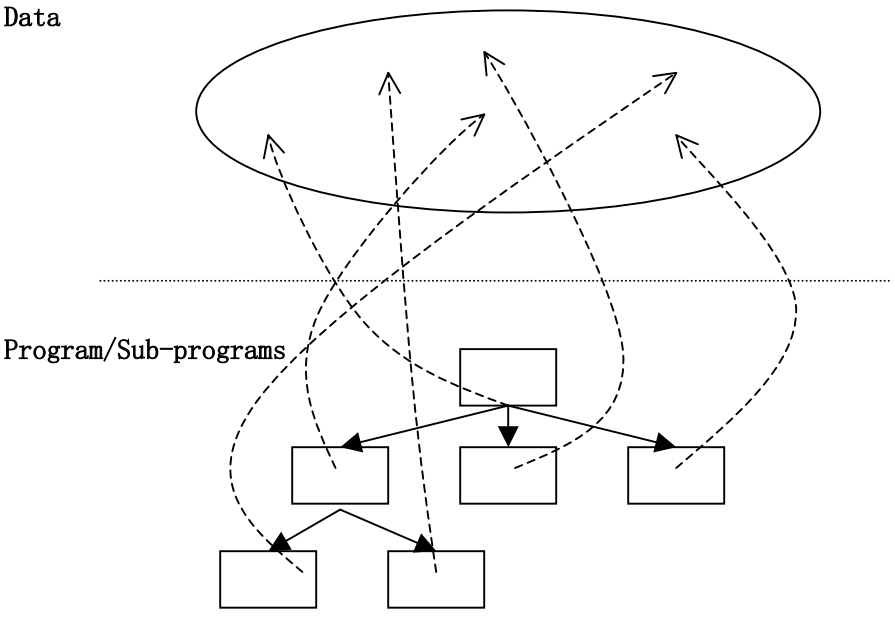
目前存在若干种程序设计风范，典型的程序设计风范有：过程式、对象式、函数式以及逻辑式等。

1、过程式

过程式程序设计是一种以功能为中心、基于功能分解的程序设计风范。在这种程序设计风范中，首先从程序的功能出发对系统的功能进行分解，然后再对分解出的各个模块进行算法和数据结构的设计。下面给出的公式刻划了过程式程序设计的本质特征：

程序 = 算法 + 数据结构

过程式程序设计的优点在于：它对程序功能的描述比较清晰。不足之处是：数据与操作分离，缺乏数据保护（图 1.3）；程序难以复用；不能适应功能的变化。

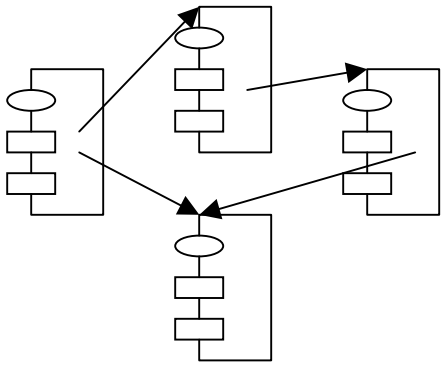


2、对象式

对象式程序设计是一种以数据为中心、基于数据抽象的程序设计风范。在对象式程序设计中，把数据及其操作作为一个整体构成对象，对象的特征由相应的类来刻划。

程序 = 对象 + 对象 + ...
对象 = 数据 + 操作

Objects



优点：加强数据保护（数据抽象与封装）；更好地支持程序复用；能够适应软件功能的变化。

缺点：程序的整体功能描述不明显，较多的信息冗余，程序效率有时不高。

3、函数式与逻辑式

函数式程序设计是基于函数及函数应用（Function Application），它基于递归函数理论和 lambda 演算，函数也作为值看待。逻辑程序设计是基于谓词演算(Predicate Calculus)，程序由一组事实和一组推理规则构成。它们常用于人工智能领域的程序开发。

1.2.2 程序设计语言

程序设计的结果必然要用一种语言表示出来，根据与计算机指令系统和自然语言接近的程度可把程序语言分为：低级语言和高级语言。

1、低级语言

低级语言是指计算机能够直接理解的语言或与之直接对应的语言，如：机器语言和汇编语言，早期的程序设计大都采用低级语言来进行。低级语言的优点在于：写出的程序效率比较高，包括执行速度快和占用空间少。其缺点是：程序难以设计、理解与维护，

难以保证程序正确性，另外，低级语言程序难以从一种型号的计算机拿到（移植）另一种型号的计算机上运行，这是因为不同型号计算机的指令系统是由差别的。

2、高级语言

高级语言是指人容易理解和有利于人对解题过程进行描述的程序语言，通常称为程序设计语言。高级语言程序须翻译成机器语言程序才能在计算机上运行，翻译方式有两种：编译与解释。编译是指把高级语言程序（称为源程序）一次性地翻译成功能上等价的机器语言程序（称为目标程序），目标程序的执行中不再需要源程序；解释是指对源程序一边翻译一边执行，这种翻译方式不产生目标程序。

高级语言的优点在于：程序容易设计、理解与维护，容易保证程序正确性。特别地，用高级语言写的程序与所采用的具体计算机的指令系统无关，因此，容易把它们移植到其它不同型号的计算机中执行，当然，目标计算机中必须要有相应语言的编译或解释程序。高级语言的缺点是：程序效率低。

程序的效率对于早期的计算机是非常重要的，早期的计算机硬件速度慢、存贮空间小，程序的效率必须通过对程序精雕细琢来提高。由于早期的计算机应用面窄、复杂度低，

典型的高级语言有：Fortran、Cobol、Basic、Pascal、C、Ada、Modula-2、Lisp、Prolog、Simula、Smalltalk、C++、Java 等。从不同的角度，可对这些语言进行分类，例如：按照应用类型，可分为科学计算语言、商务处理语言、系统程序序言等；按照所支持的程序设计风范，可分为：过程式语言、对象式语言、函数式语言、逻辑式语言以及混合式语言等。

3、程序语言的发展趋势

程序语言作为表达解题过程的工具，往往也规定了解决问题的方式。虽然高级语言比低级语言更容易描述解题过程，但目前的高级语言只是在抽象级上比低级语言略微高了一些而已，它们大都仍然是基于冯•诺依曼计算机的计算模型的，采用这些语言还必须按照计算机解题问题的方式来描述解题过程，程序设计仍然非常困难。因此，人们还在努力让计算机能够理解自然语言，使得人能够以自然语言来设计计算机程序。

1.2.3 程序设计的步骤

程序设计一般遵循以下步骤：

1、明确问题

2、确定数据结构和算法

3、用某种语言进行编程

4、测试与调试

1.3 C++语言

1.3.1 特点

C++是一个高级语言，它由 C 语言发展而来，是 C 语言的超集。C++语言在 C 语言的基础上增加了支持面向对象程序设计的语言成分，它支持过程式和对象式程序设计，属于一种混合语言。C++语言具有以下特点：

优点：灵活、高效。
缺点：不易把握，对使用者的要求较高。

1.3.2 C++程序结构

1、一个简单的 C++程序

```
//This is a C++ program
#include <iostream.h>
void main()
{ double x,y;
   cout << “Enter two float numbers:”;
   cin >> x >> y;
   double z=x+y;
   cout << "x + y =" << z << endl;
}
```

运行结果为：

```
Enter two float numbers: 7.2   9.3
x + y = 16.5
```

2、C++程序的组成

逻辑上，一个 C++程序由一些函数（子程序）、类、全局变量/对象的定义构成，其中必须有且仅有一个名字为 main 的函数，整个程序从函数 main 开始执行。函数由函数名、参数和返回类型、局部变量/对象的定义以及语句序列构成；类由数据成员和成员函数构成，函数间可以互相调用（main 除外）。变量或对象的定义可以出现在函数的外

部和内部，而语句只能出现在函数内部。

物理上，一个 C++程序可以放在一个或多个源文件（模块）中，每个文件包含一些函数、类和外部变量或对象的定义，其中有且仅有一个文件中包含一个函数 main。每个源文件可以分别编译。

1.3.3 C++语言的词法

一个语言包括语法、语义和语用三个方面。语法指程序的书写规则；语义指程序的含义；语用是指语言成分的使用场合及所产生的效果。

语法包括词法与句法，词法是指语言的构词规则，句法是指由词构成句子的规则。下面首先介绍 C++的词法（构词规则）。

1、字符集

(1) 大小写英文字母

a~z, A~Z

(2) 数字

0~9

(3) 特殊字符

空格 ! # % ^ & * _(下划线) - + = ~ < > / \ |
. , : ; ?‘ “ ( ) [ ] { }

2、单词及词法规则

单词由字符集中的字符按照一定规则构成的具有一定意义的最小语法单位。

(1) 标识符

由大小写英文字母、数字以及下划线构成，第一个字符不能是数字。如：student、student_name、x_1、_name1 等。

标识符通常用来给程序中的元素命名，如：变量名、函数名、类名、常量名、对象名、标号名、类型名等。

注意：

a)长度任意，但具体实现（编译系统）往往有所限制。

b)大小写字母有区别，如：abc、Abc 与 ABC 是不同的标识符。

c)尽量使用有意义的单词，如：age、length 等。

d)下述的关键词不能作为用户自定义的标识符，它们有特殊的作用。

(2) 关键词

语言预定义的标识符，它们有固定的含义，在程序中用作不同的目的。表 1.1 列出了 C++中的关键词。

asm[1]	**auto**	**bad_cast**	**bad_typeid**
bool	**break**	**case**	**catch**
char	**class**	**const**	**const_cast**
continue	**default**	**delete**	**do**
double	**dynamic_cast**	**else**	**enum**
except	**explicit**	**extern**	**false**
finally	**float**	**for**	**friend**
goto	**if**	**inline**	**int**
long	**mutable**	**namespace**	**new**
operator	**private**	**protected**	**public**
register	**reinterpret_cast**	**return**	**short**
signed	**sizeof**	**static**	**static_cast**
struct	**switch**	**template**	**this**
throw	**true**	**try**	**type_info**
typedef	**typeid**	**typename**	**union**
unsigned	**using**	**virtual**	**void**
volatile	**while**		

表 1.1 C++关键词表

(3) 运算符

运算符用于计算。如：+，-，*，/，>，<，==，!=，>=,<=，||，&&等。

(4) 分隔符（标点符号）

分隔符用于单词的分割。如：空格、逗号、分号、冒号、{、}等。

(5) 字面常量（直接量 **literal**）

字面常量用于表示程序中的常量。如：数值、字符及字符串等。

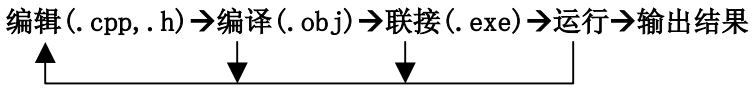
(6) 注释符

注释是为了方便程序的理解而加在程序中的文字信息。注释是给人理解程序用的，它们不构成可执行程序的一部分。C++提供了两种书写注释的方法:

a）单行注释：从符号‘//’开始到本行结束。
b）多行注释：以符号‘/*’开始到符号‘*/’结束。

1.3.4 C++程序开发环境

1、C++程序的执行过程



2、集成环境

Visual C++, Turbo-C++, Borland C++, C++ Builder，等等。

第2章 基本数据类型和表达式

数据是程序的重要组成成分，用计算机解决各种实际问题通常都是通过用计算机程序对反映实际问题的一些数据进行处理来实现的。为了能在程序中对数据进行很好地处理，首先必须对所要处理的数据的特性进行刻划，数据的特性包括：数据的结构和可施于数据的操作（运算）。在程序设计语言中通过提供数据类型机制来描述程序中的数据。

一种数据类型由两个集合构成：值集和操作（运算）集。值集描述了该数据类型包含哪些值；操作集描述了对值集中的值能实施哪些运算。如：整型就是一种数据类型，它的值集就是所有整数所构成的集合，它的操作集包括：加、减、乘、除等运算。由于受到计算机存贮空间的限制，程序设计语言所提供的数据类型的值集往往是一个有限集。

C++提供的数据类型包括基本数据类型和构造数据类型（复合数据类型）。本章介绍基本数据类型及其操作。

2.1 基本数据类型

基本数据类型往往对应着计算机能直接表示(机器指令能直接操作)的数据类型，C++提供了如下的基本数据类型：

1、整型（int）
2、字符型（char）
3、浮点型：单精度（float)、双精度（double）
4、逻辑型（bool）

对于 int、char 和 double 类型，还可以加上一些修饰符构成其它基本数据类型：

1 int：short,long,signed,unsigned
2 char：signed,unsigned
3 double: long

各种数据类型的数据需要占用一定的内存空间。在同一种规格（16 位、32 位等）的计算机上，各种不同类型的数据所占用的内存空间可能是不一样的；在不同规格的计算机上，同一种类型的数据所占用的内存空间也有可能不一样。

例如：对于字长为 32 位的计算机，一个 int 型的数据占用 4 个字节的内存空间；一个 char 型的数据占用 1 个字节的空间；一个 float 型的数据占用 4 个字节；一个 double 型的数据占用 8 个字节，等等。而对于字长为 16 位的计算机，一个 int 型的数据占用 2 个字节的内存空间。

可以用 sizeof(类型名) 来计算一台计算机上的某数据类型的数据所占的内存空间。

2.2 常量与变量

在程序中，数据通常以两种形式存在：常量和变量。

2.2.1 常量

在程序执行过程中值不能被改变的量称为常量。C++提供了：整形常量、浮点型常量、字符常量、字符串常量、枚举常量以及指针常量。

在 C++程序中,常量可以以两种形式存在，即：字面常量和有名常量（常量定义）。

1、字面常量

字面常量是指在程序中直接写出常量值的常量，通常又称为直接量（literal）。C++的字面常量有：

(1) 整形常量

整形常量可以用十进制、八进制和十六进制表示：

a）十进制表示由 0~9 数字组成，第一个数字不能是 0，如：

59，128，-72 为十进制表示；

b）八进制表示由数字 0 打头，0~7 数字组成，如：

073，0200，-0110 为八进制表示；

c）十六进制表示由 0x 或 0X 打头，0~9 数字和 A~F（或 a~f）字母组成，如：

0x3B, 0x80, -0x48，为十六进制表示

可在整型常量的后面加上 l 或 L，表示长整型常量，如：32765L；也可在整型常量的后面加上 u 或 U，表示无符号整型常量，如：4352U；也可以在整型常量的后面同时加上 u(U)和 l(L)，表示无符号长整型常量，如：41152UL 或 41152LU。

如果整型常量后面既没有 l(L)，又没有 u(U)，则为 int 型。

(2) 浮点型常量

浮点型常量由整数部分和小数部分构成，为十进制数，浮点型常量有两种表示法：小数表示法和科学表示法。

a) 小数表示法：4.07, 5., .25

b) 科学表示法：在小数表示法后加上 E(e)和一个整数，表示指数，如：3.2E-5, 5.7e10。

默认情况下，浮点型常量为 double 型。可在浮点型常量后面加上 F(f)以表示 float 型，也可在后面加上 L(l)表示 long double 型。

(3) 字符常量

由一对单引号括起来的一个字符，如：'A', '+'。引号内的字符可以是字符本身，也可以是字符的编码，如果是字符编码，必须用转义序列表示：

a) 八进制：' \ddd'，ddd 为一个八进制数。

b) 十六进制：' \xhh'，hh 为一个十六进制数。

c) 特殊表示：'\n'（换行)、'\r'（回车)、'\t'（制表)、'\b'（退格）等。

下面是字符 A 的三种表示：

' A' ⇔ ' \101' ⇔ ' \x41'

控制字符（不可打印字符）通常用转义序列表示。另外，对于下面字符的表示应特别注意：

a) \ ：'\\'

b) ' ：'\''

c) " ：'\"'或'"'

(4) 字符串常量

由双引号括起来的字符序列。如：

" This is a string"
" I' m a student"
" Please enter \" Y\" or \" N\" : "

字符串中可以包括：空格符、转义字符或其它字符。

字符常量与字符串常量的区别：

a）单引号与双引号

b）在内存中占用的字节数

c）操作

2、有名常量

首先给常量取一个名字和指定一个值，另外还可以指定一个类型。在程序中可以通过常量名来使用这些常量。

(1) 有名常量的定义

格式：

```
const <类型名> <常量名>=<值>;
        或，
#define <常量名> <值>
```

上述的常量用标识符来表示，不能是C++的关键词。

例如：

```
const float pi=3.1415926;
```

或

```
#define pi 3.1415926
```

(2) 有名常量的好处

增加可读性、增强可维护性、简化书写等。

2.2.2 变量

在程序执行中值可以被改变的量，用于保存临时性数据。

1、变量的三要素

一个变量包含三个方面的内容：名、类型和值。

(1) 变量名

变量名用于标识不同的变量。变量一般通过它们的名字（变量名）来使用。变量名用标识符表示，不能是关键词。

(2) 变量类型

在 C++中，每个变量都必须指定类型。一个变量的类型决定了该变量能取何种值、所需内存空间以及能进行何种运算（操作）。

(3) 变量的值

变量的值通常指保存在变量中的值。

变量实际上是内存空间的一个抽象，它对应着内存的一块存储区域，该内存区域有一个地址。

在 C++中往往区分变量的两种值：变量的内存地址（左值）和变量对应的内存中保存的内容（右值）。使用变量时，有时用的是变量的左值，有时用的是变量的右值。常量通常只有右值。

如：x = y + 1; 其中 x 和 y 是两个变量，x 用的是左值，y 用的是右值。

2、变量的定义与声明

程序中使用到的每一个变量都要有定义。变量定义的形式为：

〈类型名〉〈变量名表〉;

如：int a,b,c;
double x,y,z;

一个 C++程序可以由几个源文件构成，如果在一个源文件中使用到在另外一个源文件中或在本文件中使用点之后定义的变量，则在使用前需要对变量进行声明。变量声明的形式为：

extern〈类型名〉〈变量名表〉;

变量定义与声明的区别是：

(1) 变量定义要给变量分配空间。

(2) 变量定义可以给变量赋初值（初始化变量）。如：

int a=1,b=2,c=3;

(3) 在整个程序中，变量定义只能有一个，而声明可以有多个。

2.2.3 变量值的输入

变量值的来源可以有多种途径，可以是程序执行中产生的计算结果通过赋值运算得到，也可以从输入设备（如：键盘、磁盘等）中输入。下面介绍从键盘输入数据到变量

中。

C++提供了多种从键盘输入数据的途径，最典型的途径是利用抽取操作符“>>”和对象 cin 来实现，如：

```
#include <iostream.h> //插入一些用于输入/输出操作所需的声明
int i;
double d;
......
cin >> i;
cin >> d;
```

或

```
cin >> i >> d;
```

在输入时，用空格或回车符作为数据之间的分隔符，每一个输入数据的格式应与相应变量的类型相符。

例：

如果输入的数据为:12 3.4，则上述变量 i 的值为 12，变量 d 的值为 3.4。如果输入的数据为: 012 3.4，则上述变量 i 的值为 10，变量 d 的值为 3.4。如果输入的数据为:12a3.4，则上述变量 i 的值为 12，变量 d 的值没有意义。

2.3 运算符(操作符)

运算符用于对数据进行运算。C++提供了丰富的运算符，下面按功能对它们分别进行介绍。

1、算术运算符

算术运算符实现通常意义下的数值计算。分为单目运算符（一个操作数）和双目运算符（两个操作数），其中的操作数可以是常量和变量，也可以是另一个运算的结果。单目运算符有：-（取负），+（取正），--（减 1），++（增 1）。双目运算符有：+（加），-（减），*（乘），/（除），%（取余数）

注：

(1) 双目运算符一般要求两个同类型的操作数，如果不同，则进行转换，转换原则是：精度低的往精度高的转。

(2) +, -, *, / 可用于 int, float, double 和 char 类型的数据，--, ++, %只用于 int 和 char 类型的数据。

(3) / 用于两个整型数时是整除。

(4) 除了--和++外，其它操作符不改变操作数的值，只是通过相应的运算得到一个值，这个值放在一个临时的内存区域中，这个内存区域的大小与操作数中所需内存区域最大的一个相同。

--和++的操作数通常是变量，它们可以放在操作数的前面，也可以放在操作数的后面，下面以++为例介绍这两个运算符：

++x 和 x++都是把 x 的值加 1，不同的是：++x 计算得到的值是 x 加 1 以后的值；而 x++的值是 x 加 1 以前的值。如：

```
int x = 1;
y = ++x; //x 的值为 2，y 的值是 2
y = x++; //x 的值为 2，y 的值是 1
```

2、关系运算符

关系运算符用于比较两个操作数的大小，均为双目操作符。有：

>, <, >=, <=, ==, !=

结果为：true 或 false，1 或 0。其中的 true 和 false 通常称为符号常量。

操作数的类型可以是：int，float，double 和 char 等。

关系运算通常用于表示条件，如：

```
if (a > b)
  x = a;
else
  x = b;
```

3、逻辑运算符

实现逻辑运算，操作数一般为关系运算所得到的结果。有三个逻辑运算符：

(1) !（逻辑反，单目）

结果为：true 或 false，1 或 0。

```
!true -> false
!false -> true

!0 -> 1
!<非零整数> -> 0
```

(2) &&（逻辑与）

结果为：true 或 false，1 或 0。

```
false && false -> false
false && true -> false
true && false -> false
true && true -> true

0 && 0 -> 0
0 && <非零整数> -> 0
<非零整数> && 0 -> 0
<非零整数> && <非零整数> -> 1
```

(3) ||（逻辑或）

结果为：true 或 false，1 或 0。

```
false || false -> false
false || true -> true
true || false -> true
true || true -> true

0 || 0 -> 0
0 || <非零整数> -> 1
<非零整数> || 0 -> 1
<非零整数> || <非零整数> -> 1
```

(4) 短路求值（short circuit）

对于&&和||，如果第一个操作数已能确定运算结果了，则不再计算第二个操作数的值。如：

```
true || x -> true
false && x -> false
```

逻辑运算通常用于表示复合条件，如：

```
if (a == b && b == c)
    ...
```

4、位操作运算符

对整型和字符型数据按二进制位进行运算，有逻辑位运算和移位运算。

(1) 逻辑位运算

~（按位取反），&（按位与），|（按位或），^（按位异或）

异或:
0 ^ 0 -> 0
1 ^ 1 -> 0
0 ^ 1 -> 1
1 ^ 0 -> 1

(2) 移位运算

<<（左移），>>（右移）

左移：最高位舍弃，最低位补 0。
右移：

a）无符号数：最低位舍弃，最高位补 0。

b）有符号数：最低位舍弃，最高位与原来的最高位相同。

5、赋值运算符

(1) 简单赋值运算符

除了通过从设备输入以外，给变量一个值的另一种途径是通过赋值运算把一个值保存到变量中。

最简单的赋值运算是“=”，它要求两个操作数：第一个（左边）一般是一个变量，第二个（右边）是一个表达式（常量和变量是其特例），如：

a = b+c*d

赋值运算的含义是：先计算右边操作数的值，然后把计算结果赋值给（保存到）左边的变量中。

在大多数程序设计语言中，不把赋值操作作为一个运算符来看待，而是作为一个语句（赋值语句）。在 C++中，赋值是一种运算，该运算将改变第一个（左边）操作数的值，运算结果为左边的操作数。下面的计算式是一个合法的 C++表达式：

(a=b+c)*d

(2) 复合赋值运算符

为了简化书写和便于编译程序优化翻译结果，C++提供了下面的复合赋值运算符：

+=，-=，*=，/=，%=，&=，|=，^=，<<=，>>=

如果#代表上面的复合运算符，则 a #= b 表示 a = a # b。

6、其它运算符

(1) 三目运算符

C++提供了一个三目运算符：“？：”，其格式为：d1?d2:d3，d1、d2、d3 是操作数，含义是：如果 d1 的值为 true 或非零整数，则运算结果为 d2，否则为 d3。

如：x = a>b?a:b，x 的值为 a 和 b 中的最大者。

(2) 逗号运算符

可以用逗号把若干个计算式连接起来构成一个计算式，如：d1,d2,d3。含义是：先计算第一个式子，再计算第二个式子，...，整个计算式的结果为最后一个计算式的结果。

例如：x = a + b，y = c + d，z = x + y

上述的计算式等价于：z = a + b + c + d，但逻辑上比较清晰。

(3) sizeof

计算某类型的数据占用的内存大小（字节数）。格式为：

sizeof(<类型名>)，或 sizeof(<表达式>)

如：
int x;
sizeof(int)或 sizeof(x) 在 32 位的计算机上，结果为 4

(4) 强制类型转换

在进行两个操作数的运算时，如果两个操作数的类型不一样，编译系统会进行自动转换（隐式转换）。在有些不进行隐式转换的情况下，有时需要强制类型转换（显式转换），即在程序中显示地指出转换要求，其格式是：
<类型名>（<表达式>）

或

(<类型名>)〈表达式〉

例如：如果 x 和 y 是两个整型（int）变量，则在 x 与 y 的乘积超出计算机所能表示的整数范围时，x*y 将得不到正确的结果。这时应采用强制类型转换把 x 或 y 的值强制转换成精度较高的类型，如：(long)x*y 或 x*(long)y。

注意：

a) 从精度高的值强制转到精度低的值有时会丢失精度。

b) 强制类型转换是临时性的，不会改变操作数的值。

(5) 取地址(&)和取内容(*)

2.4 表达式

由运算符和操作数组成的计算式子。除了 2.3 中给出的运算符外，C++还提供了其它的运算符，其中包括圆括号（和）。操作数可以是常量、变量、函数调用以及其它一些标识符。如下面就是一个表达式：

(a+b)*c/12-max(a,b)

当连续出现两个运算符时，有时需要用空格符分开它们，例如：a+++b 有两种解释：a+ ++b 或 a++ +b，这时必须明确指出是哪一种，否则各个编译器将按照各自的规定来解释。

2.4.1 表达式的分类

根据表达式中的运算符的种类可把表达式分成：

1、算术表达式：a+5.2/3.0-9%5

2、关系表达式: (a+b*c)>d

3、逻辑表达式: a>b && c>d

4、赋值表达式: a = b+c、a += b*c+d 和 a=b=c=1

5、条件表达式: a>b?a:b

6、逗号表达式: x = a + b，y = c + d，z = x + y

7、混合表达式：(a>b)+(a=b+c)*(x>y?x:y)

2.4.2 运算符的优先级和结合性

运算符的优先级和结合性决定表达式中各个运算符的运算次序。

1、优先级

***重要性。如：x << a>b?a:b**
***记忆：一般情况**

(1) 单目，双目，三目，赋值

(2) 算术，移位，关系，逻辑位，逻辑

Symbol	**Name or Meaning**	**Associativity**
	Highest Precedence	
++	Post-increment	Left to right
−−	Post-decrement	
( )	Function call	
[ ]	Array element	
−>	Pointer to structure member	
.	Structure or union member	
++	Pre-increment	Right to left
−−	Pre-decrement	
!	Logical NOT	
~	Bitwise NOT	
−	Unary minus	
+	Unary plus	
&	Address	
*	Indirection	
sizeof	Size in bytes	
new	Allocate program memory	

delete	Deallocate program memory	
(type)	Type cast [for example, (float) i]	
.*	Pointer to member (objects)	Left to right
−>*	Pointer to member (pointers)	
*	Multiply	Left to right
/	Divide	
%	Remainder	
+	Add	Left to right
−	Subtract	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
==	Equal	Left to right
!=	Not equal	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Conditional	Right to left
=	Assignment	Right to left
*=, /=, %=, +=, −=,	Compound assignment	

<<=, >>=, &=, ^=, |=		
,	Comma	Left to right
	Lowest Precedence	

2、结合性

对于两个具有相同优先级的运算符，一般按从左到右次序，有时也按从右到左，如：单目、三目和赋值运算符。

```
a = b = c = 1
```

注意：运算符的优先级和结合性只是规定了相邻的两个运算符的运算次序，对于不相邻的运算符并没有给出规定，如对于(a+b)*(c+d)，C++并没有规定是先计算 a+b 还是 c+d。当然，对于一个双目运算符，如果其两个操作数没有副作用，则随便先计算哪一个都行。

2.4.3 表达式中的类型转换

1、单个运算的转换

对于 a 〈op〉 b，如果 a 与 b 的类型不同，则需转换。

(1) 隐式转换

```
int→unsigned→long→unsigned long→double→long double
 ↑                                  ↑
short,char                        float
```

*赋值运算和函数返回的转换例外。

(2) 显式转换

在程序中用强制类型转换实现。

2、多个运算的转换

当一个表达式中有不同类型的数据进行多个运算，则逐个运算进行类型转换。

```
int a,b;
double c;
```

a*b/c 的值有时计算不对，改成：

(double)a*b/c，或

a*(double)b/c

2.4.4 表达式的输出

表达式的计算结果除了可以通过赋值运算保存到变量中外，还可以输出到外部设备，如：显示器、打印机以及磁盘，等等，下面介绍把表达式的计算结果输出到显示器。

C++提供了多种把计算结果输出到显示器的途径，最典型的途径是利用插入操作符“<<”和对象 cout 来实现，如：

```
#include <iostream.h>
......
cout << a+b*c;
cout << a;
cout << b;
```

或

```
cout << a+b*c << a << b;
```

第3章 程序流程控制——语句

3.1 概述

表达式构成了程序的基本计算单位，当有多个表达式时，就会面临：

1、先计算哪一个表达式，或

2、根据不同的情况计算不同的表达式，或

3、其中的一个或几个表达式需要重复计算多次，等等。

对于上述问题，在程序中用语句来进行控制。程序可以包含一个或多个语句，语句描述了对程序执行流程的控制。一般情况下，语句根据它们的书写次序依次执行，如果要改变执行次序，可以用分支和循环语句来实现复杂的流程控制。语句分为单语句与复合语句。

单语句包括：表达式语句、选择语句、循环语句、转向语句以及空语句。

复合语句由用花括号({、})括起来的若干条语句构成，又称为块或分程序。复合语句中可以包含单语句、复合语句以及变量和常量的定义。在语法上，复合语句作为一条语句看待。

3.2 表达式语句

表达式加上分号“;”就可以构成语句，称为表达式语句。如：

```
a + b + c;
a = 3 * b;
x = a | b & c;
a > b ? a++: b++;
```

较常见的表达式语句是赋值、自增/自减以及输入/输出语句。如：

```
x = a+b;
x++;
cin >> a;
cout << b;
```

3.3 选择语句

根据一个条件或一个整型表达式的值从一组语句中选择一个执行。

3.3.1 条件语句 if

1、格式

if 语句有两种格式：

a) if (<表达式>) <语句>

b) if (<表达式>) <语句 1> else <语句 2>

其中，<表达式>为任意表达式，通常为关系和逻辑表达式。<语句>、<语句 1>、<语句 2>可以是一条单语句或复合语句。

2、含义

格式 a)的条件语句的含义是：如果〈表达式〉的值为 true 或非零，则执行〈语句〉，否则，执行该条件语句的下一条语句。

格式 b)的条件语句的含义是：如果〈表达式〉的值为 true 或非零，则执行〈语句 1〉，否则，执行<语句 2>。

如：.....

```
if (a > b)
  x = a;
else
  x = b;
.....
```

或

```
.....
x = b;
if (a > b) x = a;
.....
```

写条件语句时，为了提高程序的可读性，最好采用锯齿格式。

3、歧义问题

语句“if (<表达式 1>) if (<表达式 2>) <语句 1> else <语句 2>”的含义是什么？

上述语句有两种解释：

a) if (<表达式 1>) if (<表达式 2>) <语句 1> else <语句 2>

b) if (<表达式 1>) if (<表达式 2>) <语句 1> else <语句 2>

C++规定按 b)解释，即：else 子句与最近的 if 配对。可以用复合语句实现 a)的解释：

if（<表达式 1>）{ if（<表达式 2>）<语句 1> } else <语句 2>

4、例子：从键盘输入一个三角形的三条边，判断其为何种三角形。

```
#include <iostream.h>
void main()
{ int a,b,c;
   cin >> a >> b >> c;
   if (a+b <= c || b+c <= a || c+a <= b)
      cout << "不是三角形";
   else if (a == b && b == c)
      cout << "等边三角形";
   else if (a == b || b == c || c == a)
      cout << "等腰三角形";
   else if (a*a+b*b == c*c || b*b+c*c == a*a || c*c+a*a == b*b)
      cout << "直角三角形";
   else
      cout << "其它三角形";
}
```

3.3.2 开关语句 switch

1、格式

```
switch (<整型表达式>)
{ case <整型常量表达式 1>: <语句序列 1>
   case <整型常量表达式 2>: <语句序列 2>
            :
   case <整型常量表达式 n>: <语句序列 n>
   [default: <语句序列 n+1>]
}
```

其中，整型包括：int 和 char；<整型常量表达式>中不能有变量，各个<整型常量表达式>的值不能相同；〈语句序列〉由零个或多个语句构成，通常，每一个〈语句序列〉中的最后一个语句是 break 语句。

2、含义

开关语句的含义是：先计算<整型表达式>的值，如果有与之相等的<整型常量表达式 i>，则执行其后面的<语句序列 i>；否则，如果有 default 分支，则执行 default 后面的<语句序列 n＋1>，否则什么都不做。

注意：如果<语句序列 i>中没有 break 语句，则执行完<语句序列 i>后，继续执行<语句序列 i+1>。

3、例子

从键盘输入一个星期的某一天（0：星期天；1：星期一；...），然后输出其对应的英语单词。

```
.....
int day;
cin >> day;
switch (day)
{ case 0: cout << "Sunday"; break;
   case 1: cout << "Monday"; break;
   case 2: cout << "Tuesday"; break;
   case 3: cout << "Wednesday"; break;
   case 4: cout << "Thursday"; break;
   case 5: cout << "Friday"; break;
   case 6: cout << "Saturday"; break;
   default: cout << "Input error";
}
```

3.4 循环语句

根据某个条件重复执行一组语句。循环一般由四个部分组成：循环初始化、循环条件、循环体和下一次循环准备，其中的下一次循环准备包括：循环条件的改变和为下一次循环初始化。

3.4.1 while 语句

1、格式

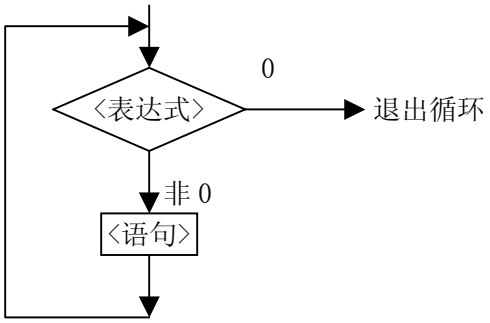
while (<表达式>) <语句>

其中，<表达式>为任意表达式，通常为关系和逻辑表达式。<语句>可以是任意的语句（单语句或复合语句）。

2、含义

<表达式>描述了循环条件，<语句>构成了循环体。while 语句的含义可以用下面的

程序流程图来解释：



while 语句显式地描述了循环条件，但循环初始化必须在 while 语句之前给出，下一次循环的准备要在循环体中给出。

3、例：求 100!

```
#include <iostream.h>
void main()
{ int i,f;
   i = 2;
   f = 1;
   while (i <= 100)
   { f *= i;
      i++;
   }
   cout << "factorial of 100 = " << f << endl;
}
```

3.4.2 do-while 语句

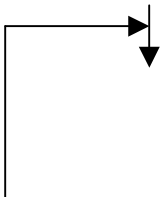
1、格式

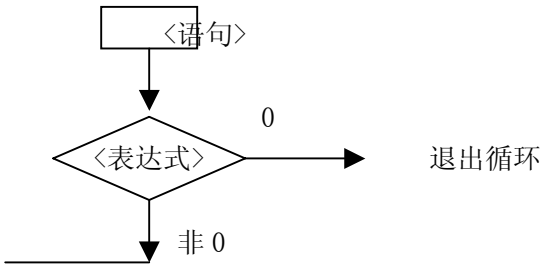
do <语句> while (<表达式>);

其中，<表达式>为任意表达式，通常为关系和逻辑表达式。<语句>可以是一条单语句或复合语句。

2、含义

do-while 语句的含义与 while 语句类似，不同之处在于：do-while 的循环体至少要执行一次。下图给出了 do-while 语句的含义：





3、例：求 100!

```
#include <iostream.h>
void main()
{ int i,f;
   i = 2;
   f = 1;
   do
   { f *= i;
      i++;
   } while (i <= 100);
   cout << "factorial of 100 = " << f << endl;
}
```

3.4.3 for 语句

1、格式

for (<表达式 1>;<表达式 2>;<表达式 3>) <语句>

其中，<表达式 1>、<表达式 2>和<表达式 3>为任意表达式，并且都可以省略。通常<表达式 1>为赋值表达式，<表达式 2>为关系或逻辑表达式，<表达式 3>为自增/自减表达式。<语句>可以是一条单语句或复合语句。

2、含义

for 循环语句的含义是：

(1) 计算<表达式 1>（循环初始化）；

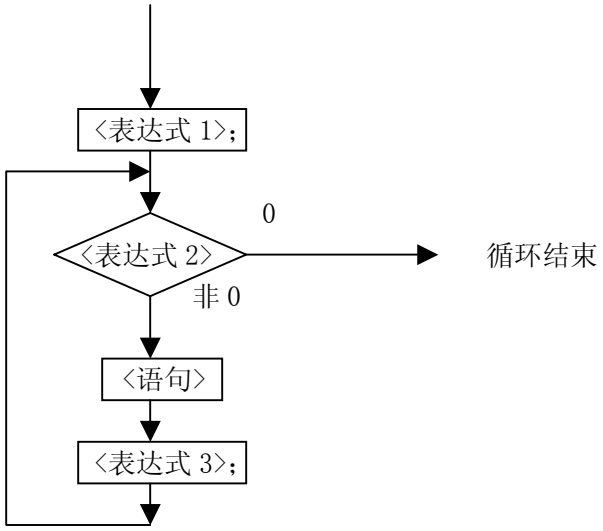
(2) 计算<表达式 2>（判断循环条件），若非零，则继续执行步骤 3）；否则，循环结束；

(3) 执行<语句>（循环体）；

(4) 计算<表达式 3>（下一次循环准备）；

(5) 转步骤（2）。

上述步骤可用程序流程图表示：



注意：<表达式 1>、<表达式 2>和<表达式 3>均可以省略。当<表达式 2>省略时，循环条件是 true 或 1。另外，<表达式 1>可以是带有初始化的变量定义，如：

```
for (int i=1; i<=10; i++)   <语句>
```

对于上述语句，变量 i 的作用范围视具体实现而定。

3、例：求 100!

```
#include <iostream.h>
void main()
{ int i,f;
   for (i=2,f=1; i<=100; i++) f *= i;
   cout << "factorial of 100 = " << f << endl;
}
```

3.4.4 while、do-while 和 for 三种循环语句的相互替代

从表达能力上讲，上述三种循环语句是等价的，它们之间可以互相替代。但是，对于某个具体的问题，用其中的某个循环结构来描述可能会显得比较自然和方便。

使用三种循环语句的一般原则是：如果循环前能确定循环的步数，或者下一次循环准备比较明确（国定），则用 for 语句；否则，使用 while 或 do-while 语句，如果循环至少执行一次，则用 do-while 语句。由于 for 语句的结构性比较好，它可以显式地表

示出：循环初始化、循环结束条件以及下一次循环准备，很多情况下都采用 for 语句。

1、while

while (<表达式>)　<语句>

1）do-while

if (<表达式>)

do <语句> while <表达式>;

2）for

for (; <表达式> ;)　<语句>

2、do-while

do <语句> while (<表达式>);

1) while

<语句>

while (<表达式>)　<语句>

2) for

<语句>

for (;<表达式>;) <语句>

3、for

for (<表达式 1>;<表达式 2>;<表达式 3>) <语句>

1) while

<表达式 1>;

while (<表达式 2>)

{ <语句>

<表达式 3>;

}

2) do-while

<表达式 1>;

if (<表达式 2>)

do { <语句> <表达式 3>; } while <表达式 2>;

3.4.5 例子

1、编程求出小于 n 的所有素数（质数）。

```
#include <iostream.h>
void main()
{ int i,j,n;
   cin >> n;
   for (i=2; i<n; i++)
   { j = 2;
      while ( j<i && i%j!=0) j++;
      if (j == i) //i 是素数
         cout << i << ",";
   }
}
```

注意：该程序效率不高，可作修改：判断 i 是否为素数，j 不必到 i-1，只需到 i 的平方根即可。

2、求第 n 个费波那契(Fibonacci)数。Fibonacci 数的定义如下：

$$fib(n) = \begin{cases} 0 & (n=1) \\ 1 & (n=2) \\ fib(n-2)+fib(n-1) & (n \geqslant 3) \end{cases}$$

```
#include <iostream.h>
void main()
{ int i,fib_n_1,fib_n_2,fib_n,n;

   cin >> n;
   fib_n_2 = 0; fib_n_1 = 1;
   for (i=3; i<=n; i++)
   { fib_n = fib_n_2 + fib_n_1;
      fib_n_2 = fib_n_1;
      fib_n_1 = fib_n;
   }
   cout << "第" << n+1 << "个费波那契数是：" << fib_n << endl;
}
```

3、求下列级数之和：

$1 + x + x^2/2! + x^3/3! + \ldots + x^n/n!$

基本想法：

用户输入两个值，分别放在变量 x 和 n 中。首先定义一个变量 sum 用于记住部分和，

初始值为 0；然后依次计算每一项的值，保存在变量 item 中，并逐个加到变量 sum 中去，其中的“依次计算...”隐含着一个循环；在计算某一项：$x^i/i!$时也隐含着两个循环，即计算 x^i 和 i!。

为了方便设计，sum 的初始值为：1+x；从第三项开始计算（循环变量 i 从 2 开始）。

程序基本结构：

```
sum = 1+x;
for (i=2; i<=n; i++)
{ item = x^i/i!;
  sum += item;
}
```

解法 1：

```
#include <iostream.h>
void main()
{ double x,sum,item,b;
  int n,a;

  cin >> x >> n;
  sum = 1+x;
  for (int i=2; i<=n; i++) //依次计算 x^2/2!、…、x^i/i!、… 、x^n/n!并加到 sum 中
  { a = 1; b = 1;
    for (int j=1; j<=i; j++)   //计算 x^i 和 i!
    { b *= x; //  计算 x^j
      a *= j; //  计算 j!
    }
    item = b/a; //  计算 x^i/i!
    sum += item; // x^i/i!加到 sum 中
  }
  cout << "x=" << x << ",n=" << n << ",sum=" << sum << endl;
}
```

解法 2：利用：$x^i = x * x^{i-1}$ 和 $i! = i * (i-1)!$，减少重复计算

```
#include <iostream.h>
void main()
{ double x,sum,item,b;
  int n,a;

  cin >> x >> n;
  sum = 1+x; b = x; a=1;
```

```
    for (int i=2; i<=n; i++) //依次计算 x^2/2!、…、x^i/i!、… 、x^n/n!并加到 sum 中
    { b *= x; // 计算 x^i
      a *= i; // 计算 i!
      item = b/a; // 计算 x^i/i!
      sum += item; // x^i/i!加到 sum 中
    }
    cout << "x=" << x << ",n=" << n << ",sum=" << sum << endl;
}
```

解法 3：利用：$item_i = item_{i-1}*x/i$ 进一步减少计算量

```
#include <iostream.h>
void main()
{ double x,sum,item;
  int n;

  cin >> x >> n;
  sum = 1+x; item = x;
  for (int i=2; i<=n; i++) //依次计算 x^2/2!、…、x^i/i!、… 、x^n/n!并加到 sum 中
  { item *= x/i; // 计算 x^i/i!
    sum += item; // x^i/i!加到 sum 中
  }
  cout << "x=" << x << ",n=" << n << ",sum=" << sum << endl;
}
```

算法 3 除了减少了计算量外，当 $x^i/i!$不太大，而 x^i或 i!很大以至于超出了计算机所能表示的数值范围时，算法 1 和算法 2 就不能得出正确的计算机结果。由于算法 3 不直接计算 x^i或 i!，而是计算 $x^i/i!$，并且计算时利用$(x^{i-1}/(i-1)!)$，因此，算法 3 有时能得出正确的计算机结果。但是，算法 3 会带来精度损失，因为 $x^i/i!$基于了 $x^{i-1}/(i-1)!$的计算结果，而 $x^{i-1}/(i-1)!$的计算结果是有精度损失的，并且这样的精度损失还会不断叠加。

3.5 转向语句

除了有条件的分支语句（if 和 switch）外，C++还提供了无条件的分支语句：goto、break 和 continue。

3.5.1 goto 语句

1、格式：

```
goto <语句标号>;
```

其中，〈语句标号〉为标识符，其定义格式为：

<语句标号>：<语句>

2、含义：

程序转至带有〈语句标号〉的语句执行。

3、例：求 100!。

```
…
int i=2,f=1;
loop: f *= i;
i++;
if (i <= 100) goto loop;
cout << "factorial of 100 = " << f << endl;
```

注意：

(1) 不能从一个语句外部转入该语句内部。

(2) 不能从一个函数外部转入该函数的内部。

3.5.2 break 语句

1、格式

```
break;
```

2、含义

(1) 退出 switch 语句

(2) 退出包含它的最内层循环语句，例如，对于求出小于 n 的所有素数的循环也可写成：

```
#include <iostream.h>
void main()
{ int i,j,n;
   cin >> n;
   for (i=3; i<n; i++)
   { for (j = 2; j<i; j++)
       if (i%j ==0) break; //跳出 for (j = 2; j<i; j++)循环
     if (j == i) //i 是素数
        cout << i << ",";
```

```
    }
}
```

3.5.3 continue 语句

1、格式：

```
continue;
```

2、含义：

立即结束本次循环，准备进入下一次循环。对于 while 和 do-while 语句，转到循环条件的判断；对于 for 语句，先计算<表达式 3>，然后计算<表达式 2>判断循环条件。

3、例子

```
while (...)
{ ......
  if (...) continue; //结束本次循环
  ......
}
```

3.5.4 关于 goto 语句

if 和 switch 属于结构化的分支语句，goto 属于非结构化的分支语句，break 和 continue 属于半结构化的分支语句。

从程序的可读性和可靠性来讲，结构化的分支语句强于半结构化的分支语句，而半结构化的分支语句又强于非结构化的分支语句。

goto 语句的缺点是：它会使得程序的静态结构和动态结构不一致、程序的可读性下降、程序容易出错。goto 语句的优点是灵活、高效。从结构化程序设计的角度讲，程序中的每一个结构都应该是单入口/单出口，goto 语句将会破坏这个规则，因此，结构化程序设计不提倡用 goto 语句。

3.6 空语句

1、格式：

```
;
```

2、含义：

不做任何事情。

3、作用：

用于在语法上需要一条语句的地方，而该地方又不需要做任何事情。

4、例子：

(1) 求出 1 至 50 之和。

```
int i,sum=0;
for (i=1; i<=50; sum+=i,i++) ;
```

(2)

```
if ((a>b || c<d) && (e>f || g<h))
   ;
else
   x++;
```

(3)

```
while (......)
{ …
   goto end; //相当于 continue
   …
   end: ;
}
```

第4章 过程抽象－－函数

4.1 子程序

4.1.1 子程序概念的产生

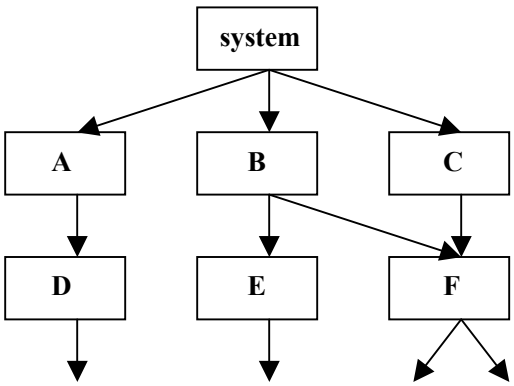
1、节省劳动力

在早期的程序设计中，人们发现程序中经常会有大量重复的代码，为了减少程序的书写量，常常把这些重复的代码从程序中抽出来，使其成为一个独立的程序单位（子程序）并为其取一个名字，程序中需要这些代码的地方用相应的名字来取代，即调用相应的子程序。

2、过程抽象

渐渐地，人们发现：子程序除了能够节省劳动力外，它有一个非常重要的作用：功能抽象，即：把一些具有独立功能的代码用子程序概念抽象出来（过程抽象），用子程序的名字来代表相应的功能。虽然在这种使用中，子程序有时只被调用一次，但它们体现了对程序功能的分解和抽象作用，使得程序容易设计和维护。

在基于过程抽象的程序设计中，首先把程序的功能分解成若干子功能，每个子功能又可以分解成若干子功能，等等，从而形成了一种自顶向下（top-down）、逐步精化（step-wise）的设计过程，其中的功能和子功能在程序中体现为：程序和子程序。



其中的抽象体现在:子程序的使用者只需要知道相应子程序的功能，而不必知道它是如何实现的。

4.1.2 参数传递机制

随着子程序概念的出现，各种参数传递机制纷纷出现。在调用子程序时，调用者通常会有数据需要传给被调用的子程序。一种数据传递方式是通过全局变量，即，调用者把要传给子程序的数据存入一些全局变量中，被调用的子程序从这些全局变量中获得调

用者提供的数据。另一种数据传递方式是通过参数传递，即调用者以参数的形式把数据传给被调用的子程序。

1、形式参数与实在参数

定义子程序时可以指明它所需要的参数，用形式参数来表示。调用子程序时，调用者将向子程序的形式参数提供数据，所提供的数据称为实在参数。

2、值传递(call-by-value)和地址传递(call-by-reference)

在调用子程序时把实在参数传给形式参数的过程称为参数传递，参数传递的方式可以有多种，较常见的参数传递方式是值传递和地址传递。

值传递是指把实在参数的值传给相应的形式参数，而地址传递是指把实在参数的地址传给相应的形式参数。值传递的长处在于它能够阻止子程序通过形式参数来改变实在参数的值，从而减少函数的副作用，其不足之处在于：当参数传递的数据量较大时，参数传递的效率不高。地址传递的优势在于能够提高参数传递的效率，其劣势在于：子程序中必须采用间接方式来访问传递的数据，这给使用带来不变。另外，地址传递也可能会导致函数的副作用，即子程序能通过形式参数来改变实在参数的值，这往往会导致程序的一些数学性质（如交换率）遭到破坏。

4.2 函数

函数是 C++提供的用于实现子程序的语言机制。

4.2.1 函数的定义

1、格式：

〈返回值类型〉〈函数名〉(〈形式参数表〉)〈函数体〉

其中，〈返回值类型〉为任意的 C++类型：基本类型和构造类型。也可以是 void，它表示函数没有返回值。也可以缺省(不是 void)，这时系统当 int 解释。

〈函数名〉为标识符

〈形式参数表〉为零个、一个或多个形参说明（用逗号隔开），

形参说明的格式为：

<类型> <形参名>

<函数体> 为一个<复合语句>。

2、return 语句

在函数体中可以包含 return 语句，用于函数执行的结束控制。

(1) 格式：

return <表达式>;

或

return;

(2) 含义：

结束函数体的执行，返回调用者。当〈返回值类型〉不为 void 时，函数体中应采用格式(1)的 return 语句，<表达式>的值作为函数的返回值（计算结果）返回给调用者，必要时，根据〈返回值类型〉进行类型转换。当〈返回值类型〉为 void 时，应采用格式(2)的 return 语句。

函数体中也可以没有 return 语句，这时〈返回值类型〉一定为 void，函数体执行完最后一条语句后返回。

例：求阶乘的的函数

```
int factorial(int n)
{ int f=1;
   for (int i=2; i<=n; i++) f*=i;
   return f;
}
```

4.2.2 函数的调用

1、格式

对于定义的一个函数，必须要调用它，它的函数体才开始执行。除了 main 函数外，程序中定义的其它函数的调用一定是从 main 中开始的。对 main 函数的调用是由系统产生的。

函数在使用前一定要定义，如果函数定义在另外一个源文件（模块）中或在本源文件的使用点之后，则在使用前需要声明。

函数声明采用函数原型来表示，格式如下：

[extern]〈返回值类型〉〈函数名〉(〈形式参数表〉);

注意：〈形式参数表〉中可以只给出形参类型而不写形参名。

函数调用是一个表达式，格式如下：

〈函数名〉(〈实在参数表〉)

其中，〈函数名〉为已定义或声明的一个函数的名字；〈实在参数表〉为零个、一个或多个实在参数（用逗号分隔），每个实在参数是一个表达式。

例：

```
#include <iostream.h>
extern int factorial(int n);
void main()
{ int x;
   cin >> x;
   cout << "Factorial of " << x << " is " << factorial(x) << endl;
}
```

函数调用也可以加上分号（;）构成语句来使用,这里的函数通常是无返回值的函数（函数返回类型为 void）。

2、函数调用过程

函数调用按以下步骤进行：

(1) 计算实参的值（对于多个实参，C++没有规定计算次序）。

(2) 把它们分别传递给被调用函数的形参，

(3) 执行函数体。

(4) 函数体中执行 return 语句返回函数调用点，调用点获得返回值（如果有返回值），进行调用以后的操作。

3、参数传递

C++提供了两种参数传递机制：值传递和地址传递。下面先介绍值传递机制，地址传递在介绍指针数据类型时给出。

在值传递机制中，实参的值通过类似于赋值运算的形式赋值给相应的形参，必要时根据形参的类型进行类型转换。函数体中对形参值的改变不会影响实参的值。

例：

```
#include <iostream.h>
double sum_double(double x, double y)
{ double z;
   z = x + y;
   return z;
}
void main()
{ double a,b;
   cin >> a >> b;
   cout << sum_double(a,b) << endl;
```

}

(1) 执行 main 时，产生两个变量（分配内存空间）a 和 b:

(2) 调用 sum_double 函数时，又产生两个变量 x 和 y:

并且，执行类似下面的语句：

```
x = a;
y = b;
```

4、函数体的执行和返回

参数传递完毕后，被调用的函数体开始执行，如果函数体中有变量定义，则函数体执行中将产生新的变量。函数体一直执行到 return 语句或最后一条语句结束，如果函数有返回值，则该返回值将存放在临时单元中，然后返回到调用处继续执行函数调用后的操作。

注意：函数体中不能用 goto 语句转向调用点或函数外的任何其它地方，也不能用 goto 语句转入函数体。

4.2.3 函数的例子

1、编程求出 50 至 100 之内的所有素数。

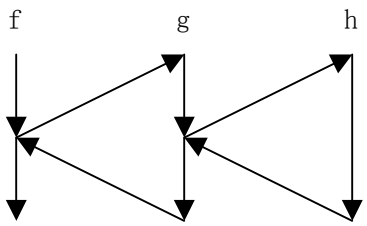
```
#include <iostream.h>
#include <math.h>
#define MIN 51
#define MAX 100
bool is_prime(int n)
{ int i,j;
   j = (int)sqrt((double)n);
   for (i=2; i<=j; i++)
      if (n%i == 0) return false;
   return true;
```

```
}
void print_prime(int n, int count)
{ if (count % 6 == 0) cout << endl;
   cout << " " << n;
}
void main()
{ int i,n=0;

   for (i=MIN; i<MAX; i+=2)
   { if (is_prime(i))
      { print_prime(i,n);
        n++;
      }
   }
   cout << endl;
}
```

4.3 递归函数

4.3.1 递归函数的定义

函数的调用是可以嵌套的，返回时将根据嵌套层次逐层返回。下图给出了函数 f 中调用函数 g，函数 g 中又调用 h，然后逐层返回的情形：



如果一个函数在其函数体中直接或间接地调用了自己，则该函数称为递归函数。

1、间接递归

```
g();
f()
{ …
   g();
```

```
    …
}
g()
{ …
   f();
   …
}
```

2、直接递归

```
f()
{ …
   f();
   …
}
```

4.3.2 递归函数的作用

循环为解决重复执行的问题提供了一种途径，这种重复执行的方式称为迭代。解决重复执行问题的另一个途径是采用递归函数调用。

有些问题用递归函数来解决会显得比较自然，特别是对于递归定义的问题，如 fibonacci 数问题：

$$fib(n)=\begin{cases}0 & (n=1)\\ 1 & (n=2)\\ fib(n-2)+fib(n-1) & (n\geqslant 3)\end{cases}$$

在 3.4.5 中用循环语句给出了是求第 100 个 fibonacci 数的迭代算法，下面给出它的递归算法：

```
#include <iostream.h>
int fib(int n)
{ if (n == 1) return 0;
   else if (n == 2) return 1;
   else return fib(n-2)+fib(n-1);
}
void main()
{ cout << fib(100) << endl;
```

```
}
```

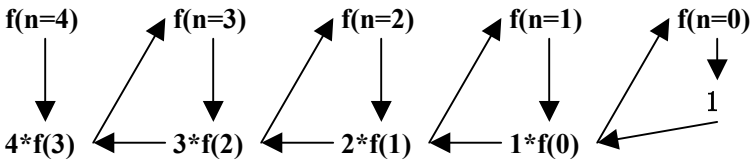
4.3.3 递归函数的执行过程

可以把一个递归函数看成有多个实例，每个实例都是一个单独的函数，然后按函数的嵌套调用来理解递归调用。

例：求 n!的递归函数。

```
int f(int n) //计算 n!
{ if (n == 0) return 1;
   else return n*f(n-1);
}
…
f(4);
```

上述递归函数调用 f(4)可以按照下面的函数嵌套调用来理解：



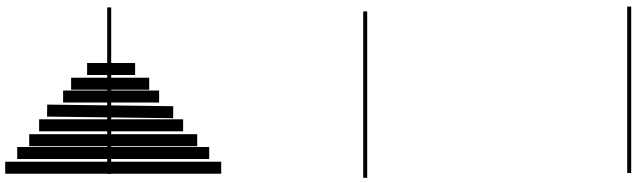
在定义递归函数时，一定要对两种情况给出描述：

(1) 递归条件：何时需要递归调用。

(2) 结束条件：何时不需递归。

4.3.4 例子

汉诺塔（Hanoi）问题：有 A，B，C 三个柱子，柱子 A 上穿有 8 个大小不同的圆盘，大盘在下，小盘在上。现借助于柱子 C 把柱子 A 上的所有圆盘以原来的叠放次序移到柱子 B 上，要求每次只能移动一个圆盘，且大盘不能放在小盘上。请编写一个 C++函数给出移动步骤，如：A→C, A→B,…。



A B C

解题思路：

(1) 把 7 个盘子从柱子 A 移到柱子 C

(2) 把第 8 个盘子从柱子 A 移到柱子 B

(3) 把 7 个盘子从柱子 C 移到柱子 B

程序如下：

```
#include <iostream.h>
void hanoi(char x, char y, char z, int n)
{ if (n == 1)
      cout << "1: " << x << "->" << y << endl;
   else
   { hanoi(x,z,y,n-1);
      cout << n << ": " << x << "->" << y << endl;
      hanoi(z,y,x,n-1);
   }
}
…
hanoi('A','B','C',8);
…
```

4.4 作用域

4.4.1 C++程序结构与作用域

逻辑上，一个 C++程序由一些函数（子程序）和一些全局（外部）变量或对象的定义构成，其中必须有且仅有一个名字为 main 的函数，整个程序从函数 main 开始执行。函数由一些语句和一些局部变量或对象的定义构成，函数间可以互相调用（main 除外）。变量或对象的定义可以出现在函数的外部（全局变量）和内部（局部变量），而语句只能出现在函数内部，函数内部不能再定义函数。

物理上，一个 C++程序可以放在一个或多个源文件（模块）中，源文件的文件名一般以.cpp 和.h 作为扩展名(类型名)，每个文件包含一些函数和全局变量或对象的定义，其中有且仅有一个文件中包含一个函数 main。每个源文件分别编译，编译后通过一个连接程序把它们以及程序中用到的标准函数所在的函数库连接成一个可执行的程序。

程序中用到的每个元素都要有定义，这里的元素包括：常量、变量、函数、类、对象以及语句标号等。程序中定义的元素往往被限制在一定的范围内使用，这个范围称为作用域。作用域是指一个定义了的标识符能被使用的范围，这里的标识符可以是常量名、变量名、函数名、类名、对象名以及语句标号等。一个标识符的作用域往往与该标识符

的定义位置有关。

在一个标识符的作用域中使用该标识符时，如果未见到该标识符的定义，则在使用前往往需要声明之。

C++的作用域主要有下面 4 种：

1、程序级：构成程序的所有源文件。

2、文件级：定义标识符的源文件。

3、函数级：定义标识符的函数。

4、块级：定义标识符的复合语句或分程序。

4.4.2 变量的作用域与生存期

1、全局变量

全局变量是指在函数外定义的变量，其作用域一般是程序级。若在定义时加上 static 修饰符，则作用域为文件级，如：

```
int x;  //程序级
static int y;  //文件级
f()
{ ......
}
```

使用全局变量时，若变量的定义在其它文件中（非 static），或在本文件中使用点之后，则在使用前需要声明它们，格式如下：

```
extern <类型> <变量>;
```

例如：

```
//file1.cpp
int x=1;
extern int y;
main()
{ ...x ... //Ok
  ...y ... //Ok
  ...z ... //Error, z 未声明
  ...c ... //Error, c 不可使用
}
```

```
//file2.cpp
int y=0;
static char c='A';
f()
{ ...y ... //Ok
  ...c ... //Ok
  ...x ... /Error, x 未声明
  ...z ... /Error, z 未声明
}
double z=2;
g()
{ ...z ... //Ok
}
```

通常，在一个源文件中定义的、允许在其它源文件中使用的全局变量，往往把它们的声明放在某个.h 文件中，在需要使用这些全局变量的源文件中写上：

```
#include "xxx.h"
```

例如：

```
//file1.cpp
#include "file2.h"
int x=1;
main()
{ ......
}

//file2.h
extern int y;
extern double z;
f();
g();

//file2.cpp
int y=0;
static char c='A';
extern double z;
f()
{ ......
}
double z=2;
```

```
g()
{ ......
}
```

一般来讲，一个模块通常由两个文件构成：.cpp 和.h，其中，.cpp 文件中包含该模块的所有函数和全局变量的定义，.h 文件中包含该模块允许上其它模块使用的函数和全局变量的声明。

2、局部变量

在复合语句内定义的变量，其作用域为从定义点开始到复合语句结束。

例如：

```
f()
{ …
   int x; //变量 x 从下一条语句开始可以使用直到函数结束。
   …
}
```

3、形式参数

形式参数的作用域为函数级，即从函数头开始到函数体结束。

在同一个作用域中不能定义两个同名的变量，即在同一个作用域中，不同的变量不能取相同的名字。在不同作用域中定义的变量可以同名，它们是不同的变量。如：

```
f(int a)
{ int x,y;
   double x; //Error: x redefined
   ......
}

g(double a) //Ok
{ double x,y; //Ok
   ......
}

h()
{ int a; //Ok
   ...
```

```
    f(a);
    ...
}
```

如果同名变量处于内外层，则在内层中使用的是内层定义的变量，如果在内层要使用与内层定义同名的全局变量，则需要加上全局分辨符“::”，如：

```
int x;
f()
{ double x;

   x //double x
   ::x //int x
}
```

4、变量的生存期

变量的生存期是指变量占用内存的时间段，或变量存在的时间段。全局变量的生存期是整个程序运行期间；局部变量和形式参数的生存期一般为函数被调用期间，但是，在定义局部变量时可以加上存贮类修饰符 static，这时，该局部变量的生存期为整个程序运行期间，如：

```
f(int a) //a 的生存期为函数 f 被调用期间
{ int x=0; // x 的生存期为函数 f 被调用期间
   static int y=1; //y 的生存期为整个程序运行期间
   x++;
   y++;
}
```

另外，定义局部变量时还可以加上 auto 或 register 存贮类修饰符。auto 为默认存贮类，表明局部变量的生存期为函数被调用期间；register 存贮类建议编译器把相应的局部变量的空间分配在寄存器中，以提高程序执行效率。

在定义一个变量时如果没有显式给出初始化，则，对于全局变量和 static 类的局部变量，系统自动把它们按位模式初始化为 0；对于非 static 类的局部变量，系统不对它们初始化。

当一个程序准备运行时，操作系统将为其分配一块内存，其中包括四个部分：全局数据区(global)、代码区(code)、栈区(stack)和堆区(heap)，下图给出了程序内存分配示意图，各个部分在内存中的次序可能因不同的操作系统会有不同：

全局数据区
代码区
栈区
堆区

其中，全局数据区用于全局变量、static 类局部变量以及常量的内存分配；代码区用于存放程序的指令，对 C++而言，代码区存放的是函数；栈区用于非 static 类的局部变量、函数的形式参数以及函数调用时的有关信息（如：函数返回地址等）的内存分配；堆区用于程序运行时动态生成的变量（动态变量）的内存分配。

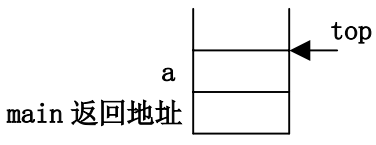
全局数据区和代码区的大小是固定的，而栈区和堆区的大小将会随着程序的运行在不断变化，不过，操作系统通常会限制栈区和堆区空间的大小。

函数的形式参数和非 static 类局部变量的内存空间是在栈区分配的，它们随着函数的调用和返回在不断地变化。

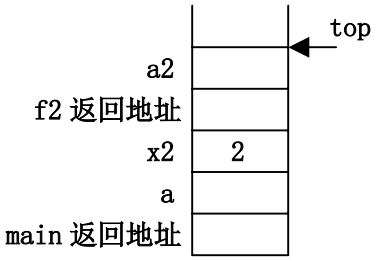
例子：下面给出了一个例子，它说明了栈空间随着函数调用的变化情况。

```
void f1(int x1)
{ int a1;
   ......
}
void f2(int x2)
{ int a2;
   ......
   f1(1);
   ......
}
void f3(int x3, int x4)
{ int a3;
   ......
}
void main()
{ int a;
   f2(2);
   f3(3,4);
}
```

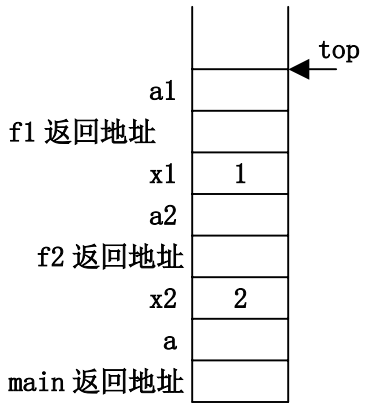
(1) 调用 **f2** 前



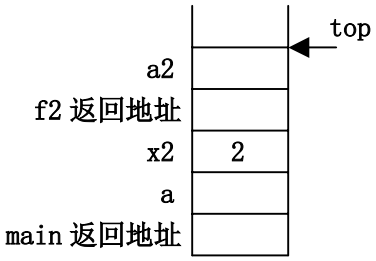
(2) 调用 f2 后、调用 f1 前



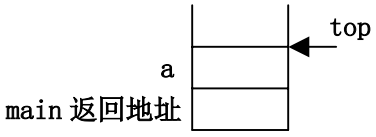
(3) 调用 f1 后、f1 返回前



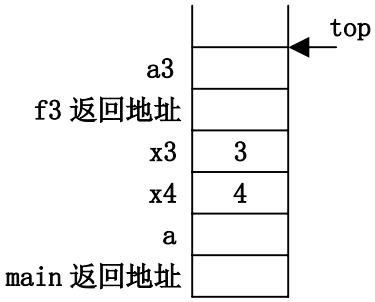
(4) f1 返回后、f2 返回前



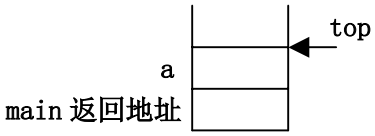
(5) f2 返回后、调用 f3 前



(6) 调用 f3 后、f3 返回前



(7) f3 返回后、main 返回前



栈空间一方面被各个函数共享，另一方面也对函数调用的深度（嵌套调用）有所限制。特别对于递归函数调用，虽然从理论上讲递归层次是任意的，但由于受到栈空间的限制，实际递归调用层次是有限制的，很多的递归函数在运行时由于递归层次太深而导致程序运行异常终止：Stack overflow。因此有些问题虽然用递归函数解决比较自然，考虑到实际的内存情况，有时不得不用迭代（循环）来解决。

4.4.3 函数的作用域

函数一般为程序级作用域，但是，如果在定义函数时加上 static 修饰符，则为文件作用域。

在函数的作用域内调用它们时，若未见到函数的定义，则需要声明之，函数声明用函数原型来给出。

4.4.4 其它标识符的作用域

1、标号的作用域

标号的作用域为标号定义所在的块。

2、常量的作用域

用 const 定义的常量的作用域为：

(1) 局部常量同局部变量。

(2) 全局常量为文件作用域。

(3) 全局常量定义时加上 **extern** 修饰符则为程序作用域。

用#define 定义的常量的作用域为定义点开始到文件结束或遇到#undef。

4.4.5 名空间（namespace）

对于一个多文件构成的程序，有时会面临下面的问题：

1、在一个源文件中用到一个在另外一个源文件中定义的元素（如函数），而该元素的名字与本源文件中定义的一个元素的名字相同。

2、在一个源文件中用到两个分别在另外两个源文件中定义的元素，而这两个元素具有相同的名字。

为了解决上述的名冲突问题，C++提供了名空间（namespace）设施，即给一些定义或声明取一个名字，当需要使用这些定义或声明的元素时，用相应的空间名字来受限。如：

```
//file1.h
namespace A
{ void f();
  void g();
}

//file1.cpp
namespace A
{ void f() { .....}
  void g() { .....}
}

//file2.h
namespace B
{ void f();
  void g();
}

//file2.cpp
namespace B
{ void f() { .....}
  void g() { .....}
}

//main.cpp
```

```
#include "file1.h"
#include "file2.h"
void main()
{
   A::f(); //file1 中定义的 f
   A::g(); //file1 中定义的 g
   B::f(); //file2 中定义的 f
   B::g(); //file2 中定义的 g
}
```

如果在某个名空间中定义了很多元素，这时，会给使用这些元素带来不便。为了简化书写，当使用某个名空间上的元素时，可在使用前写一个 using 指示项，使得今后使用相应名空间中的元素时不必用空间名受限，如：

```
void main()
{ using namespace A;
   f(); //file1 中定义的 f
   g(); //file1 中定义的 g
   B::f(); //file2 中定义的 f
   B::g(); //file2 中定义的 g
}
```

也可以分别对某个名空间中的元素采用 using 声明，如：

```
void main()
{ using A::f;
   f(); //file1 中定义的 f
   A::g(); //file1 中定义的 g
   B::f(); //file2 中定义的 f
   B::g(); //file2 中定义的 g
}
```

4.5 带缺省值的形式参数

在定义或声明函数时，可以指定某些参数的缺省值，当调用时没有提供相应的实参，则相应的实参采用缺省值。

例：

```
void f(int a, int b=1, int c=0)
{ …
}
void main()
{ int x=0,y=1,z=2;
```

```
    f(x,y,z); //OK
    f(x,y); //OK
    f(x); //OK
    f(); //Error
}
```

注意：

(1) 带缺省值的形参应处于形参表的右部。

```
void f(int a, int b=1, int c);   //error
```

(2) 带缺省值的形参定义只在函数声明处有意义，在不同的作用域对同一个函数的缺省值可以有不同的声明。

```
//file.cpp
void f(int a, int b, int c)
{ ......
}

//file1.cpp
void f(int a, int b, int c=2);
......
f(0,1);
.......

//file2.cpp
void f(int a, int b=1, int c=2);
.......
f(0);
.......
```

4.6 内联函数

函数作为一种抽象机制，对解决大型复杂问题起到了很大的作用。但同时也带来了程序执行效率的降低，特别对频繁调用一个小函数的程序，其执行效率是不高的。

C++中解决上述问题的办法有两种：宏定义和内联函数。

1、宏定义

例：

```
#define max(a,b) a>b?a:b
…
int x,y;
…
cout << max(x,y) << endl;
```

注意：

(1) 采用宏定义时，如果一行写不下，在本行最后用续行符“\”转到下一行。

(2) 宏定义是文本替换，应加上必要的括号。

例如：用上面的 max 宏定义，下面的表达式：

```
10+max(x,y)+z
```

将替换成：

```
10+a>b?a:b+z
```

它与要求不符，因此，上面的 max 宏定义是不可靠的，应该定义为：

```
#define max(a,b) (((a)>(b))?(a):(b))
```

```
#define multiply(a,b) a*b
…
multiply(x+y,m-n)    // x+y*m-n
```

宏定义的缺点：

(1) 重复计算。

(2) 参数类型检查不足。

2、内联函数

在函数定义前加上关键字 inline，建议编译器把函数体扩展到调用点。内联函数具有宏和函数二者的优点。

注意：

(1) 何时使用内联函数？

(2) 编译器对内联函数的限制

(3) 在内联函数的调用点一定要见到内联函数的定义。

4.7 函数重载(Overloading)

1、定义

在同一个作用域中定义的多个名字相同、参数不同的函数，它是 C++提供的一种多态机制（论域中的一个元素有多种解释）。

函数重载主要用于功能相同而参数不同的多个函数的定义。

例如：

```
int add(int x, int y)
{ return x+y;
}
double add(double x, double y)
{ return x+y;
}
```

2、绑定（Binding）(定联、联编)

确定一个对重载函数的调用对应着哪一个重载函数定义的过程。

如：

```
......
int a,b;
add(a,b); //调用上述的 int add(int x, int y)
......
```

绑定的基本规则是：

(1) 精确匹配：实参与形参的个数与类型完全相同。

(2) 提升匹配：

a）char, unsigned char, short int 自动提升到 int

b）如果 unsigned short int 比 int 小，则提升到 int，否则提升到 unsigned int

c）float 提升到 double

例：

```
f(int);
f(double);
......
char ch;
```

```
float x;
f(ch); //f(int);
f(x); //f(double);
```

(3) 标准转换匹配

a）任何数值类型可以互相转换

b）枚举类型可以转换成任何数值类型

c）零与任何数值类型或指针类型匹配

d）任何类型的指针可以与 void *匹配

e）每个标准转换都是平等的。

例：

```
f(int);
f(void *p);
......
f(1.0) //f(int);
```

(4) 自定义转换匹配

根据上述匹配规则，如果无法匹配或匹配不唯一，则绑定失败。

例：

```
void f(int a);
void f(float d);
void f(char *p);
......
int x[10];
f(x); //error
f('a'); //f(int);
f(1.0); //error
```

4.8 预处理命令

C++提供了一些供编译程序使用的命令：预处理命令，这些命令不是 C++语言的一部分，而是对编译过程给出指导。

1、文件包含命令

(1) 格式：

#include <文件名>

或

#include "文件名"

(2) 含义：

用文件名所指定的文件内容替代该命令。当文件名用尖括号（<>）括起来时，它表示在系统指定的目录下寻找指定文件；当文件名用双引号括号（“”>）括起来时，它表示先在包含#include 的源文件所在的目录下查找指定文件，然后再在系统指定的目录下寻找指定文件。

用#include 包含的文件一般为头文件（.h）。对于每一个.cpp 文件，一般都有一个对应的.h 文件，在.h 文件中给出了在相应的.cpp 文件中定义的、需要在其它.cpp 文件中使用的一些内容（变量、函数等）的声明。.h 文件实际上是给出了一个模块的接口。

2、宏定义

(1) 格式：

#define <宏名> <文字串>
#define <宏名>（<参数表>） <宏体>
#define <宏名>
#undef <宏名>

(2) 含义

文字替换。

3、条件编译命令

(1) 格式：

a)

#ifdef <标识符>
<程序段 1>
[#else
<程序段 2>]
#endif

b)

#ifndef <标识符>
<程序段 1>
[#else
<程序段 2>]
#endif

c)

#if <常量表达式 1>/#ifdef/#ifndef
<程序段 1>
#elif <常量表达式 2>
<程序段 2>
…
#elif <常量表达式 n>
<程序段 n>
[#else
<程序段 n+1>]
#endif

4.9 函数库

为了方便程序设计，编译系统往往提供了标准函数库，其中包含了一些有用的函数，如常用的数学函数（指数、对数、三角等）、字符串处理函数、图形函数，等等，特别对于C++的输入/输出功能，往往也作为库函数由编译系统提供。当程序中需要这些功能时，可直接调用它们即可。

不同的编译系统所提供的库函数可能会不同，使用它们时往往需要查阅相应编译系统的用户手册。不过，对于一些常用的功能，各个编译系统往往是相同的。

编译系统往往根据库函数的功能对它们进行分类，每一类库函数的原型分别放在一个头文件（.h）中，程序中使用某个库函数时，应该在源文件中包含相应的头文件。为了程序能够运行，必须在连接(link)时，把相应库函数的代码连接到程序中来。

第5章 复合数据类型

C++除了提供基本数据类型(int,char,float,double,bool)外，还提供了用基本数据类型构造复杂数据类型的机制，使得程序能对复杂数据进行描述与处理。

5.1 枚举类型

1、定义

(1) 格式：

a) enum <枚举类型名> {<枚举表>};

b) enum {<枚举表>} <变量名表>;

其中，<枚举类型名>为标识符，它标识枚举类型的名；

<枚举表>为用逗号隔开的若干个枚举值，每个枚举值为一个符号常量，用标识符表示。

<变量名表>为用逗号隔开的若干个变量名。

例：

```
enum day { Sun,Mon,Tue,Wed,Thu,Fri,Sat };
enum color {RED,GREEN,BLUE};
```

(2) 含义

定义了一个类型，该类型的值集由枚举表中的枚举值构成，每一个枚举值对应着一个整型数，通常情况下，第一个枚举值对应常量值 0，第二个枚举值对应常量 1，依次类推。对整型的运算可以实施到枚举类型。

在定义枚举类型时，也可以给枚举值指定对应值，如：

```
enum day { Sun=7,Mon=1,Tue,Wed,Thu,Fri,Sat };
```

这时，Sun 对应 7，Mon 对应 1，Tue 对应 2，Wed 对应 3，…。

枚举类型增加了程序的可读性。

bool 类型是 C++语言提供的一个预定义的枚举类型：

```
enum bool { false, true };
```

2、枚举类型的变量定义

格式：

(1) enum day { Sun,Mon,Tue,Wed,Thu,Fri,Sat } d1,d2,d3;

(2) enum day d1,d2,d3;

(3) enum { Sun,Mon,Tue,Wed,Thu,Fri,Sat } d1,d2,d3;

(4) day d1,d2,d3;

3、运算

(1) 赋值

```
d1 = Sun;    //OK
d1 = 3;      //Error
d1 = RED;    //Error
d1 = (day)3; //OK，不提倡
```

(2) 其它运算

```
int x,y;
x+1+Sun*10+Wed //OK,不提倡

day x;
for (x=Sun; x <= Sat; x++) …

switch (d1)
{ case Sun:…
   case Mon:…
   …
}
```

5.2 数组类型

用于表示由固定多个同类型的元素按一定次序所构成的数据，如：向量，矩阵等。

格式：

<类型> <数组变量名>[<大小 1>][<大小 2>]…[<大小 n>];

其中，<类型>为任意 C++类型，<大小 1>、<大小 2>、…、<大小 n>表示数组的每一维的元素个数,为常量表达式。

例：

int a[10],b[10]; //a,b 分别为由 10 个整型数所构成的整型数组
double c[20][10]; //c 为一个二维数组，

注意：由于栈空间有限，因此，不要定义很大的局部数组变量！

5.2.1 一维数组

一维数组通常用于表示由固定多个具有线性次序的同类型数据所构成的复合数据，如：向量。

1、一维数组类型的定义

通常，一维数组类型的描述及其变量的定义是同时给出的，如：

int a[10];　　//定义了一个由 10 个整型数所构成的一维数组类型的变量 a

一维数组类型的描述也可以单独进行，这是通过采用 C++的类型定义（typedef）机制来给一维数组类型取一个名字来实现的，如：

typedef int A[10];　//定义了一个由 10 个整型数所构成的一维数组类型 A
A a;　　//定义了一个类型为 A 的一个数组变量 a

typedef 除了可以用来给一维数组类型取一个名字外，它还可以用于给其它 C++类型取名，如：

typedef unsigned int UINT;
unsigned int i;　<==>　UINT i;

2、一维数组元素的访问

int a[10];
a[0] = 1;
a[1] = 2;
…
a[9] = 10;

一般格式是：
<一维数组变量名>[<下标表达式>]

其中，〈下标表达式〉为整型表达式。

例如：

```
int i,j;
…
a[i+j] = 123;
```

注意下标越界问题：

```
a[10] = 11; //Error
```

对一维数组的每一个元素访问一次通常需要一个循环：

```
for (int i=0; i<10; i++) { … a[i]…}
```

3、一维数组的初始化

除了用赋值运算给一维数组变量进行初始化外，还可以在定义一维数组变量时进行初始化：

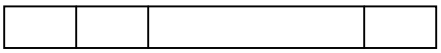
```
int a[10]={1,2,3,4,5,6,7,8,9,10};
int b[10]={1,2,3,4};
int c[]={1,2,3}; //c 有三个元素
```

4、一维数组的存贮

在内存中，一维数组变量将占用连续的存贮空间，所占空间的大小可由 sizeof 运算符计算。

```
int a[10];
```

a[0]　a[1]　...　a[9]



```
cout << sizeof(a);
```

5、向函数传递一维数组

当需要把一个一维数组传给一个函数时，实参只需写数组变量名，函数形参为不带数组大小的数组定义。

例：

```
void f(int x[],int num)
{ ......
}

void main()
{ int a[10],b[20];
  …
```

```
    f(a,10);
    …
    f(b,20);
    …
}
```

函数 f 的形参中之所以需要一个表示数组元素个数的参数 num，是因为函数 f 不知道实参数组的大小。如果传递的是字符数组，则没有必要传递数组大小（为什么？）。

注意：

数组作为函数参数传递时，实际传递的是数组的首地址，函数中对形参数组值的改变将导致实参数组值的变化！

6、字符数组（字符串）

```
char s[10];
```

一维字符数组变量通常用于存贮字符串。在定义一维字符数组的大小时，它应比实际能够存贮的字符串中最大字符个数多一个，因为在存贮字符串时，通常在最后一个字符的后面存贮一个'\0'字符（编码为 0），以表示字符串结束。

例如：

```
cin >> s;
```

系统从键盘输入一个字符串，放入 s，并在最后一个字符的后面放一个'\0'。处理这个字符串时通常采用：

```
for (i=0; s[i] != '\0'; i++) …
```

字符数组变量的初始化：

```
char s[10]={'h','e','l','l','o','\0'};
char s[10]={"hello"};
char s[10]="hello";
char s[]="hello";
```

注意：

对字符数组赋值时，只能逐个元素赋值，不能用一个字符串常量对整个数组赋值，如：

```
char s[10];
s = "ABCDEFG"; //Error
```

C++函数库中提供了一些用于对字符串进行操作的函数，主要有：

```
#include <string.h>
```

a）计算字符串的长度（字符个数）

```
int strlen(const char s[]);
```

b）字符串复制

```
char *strcpy(char dst[],const char src[]);
char *strncpy(char dst[],const char src[],int n);
```

c）字符串拼接

```
char *strcat(char dst[],const char src[]);
char *strncat(char dst[],const char src[],int n);
```

d）字符串比较

```
int    strcmp(const char s1[],const char s2[]);
int    strncmp(const char s1[],const char s2[],int n);
```

7、应用

例 1、求第 100 个费波那契(Fibonacci)数。

```
#include <iostream.h>
const int N=100;
void main()
{ int a[N],n;

   a[0] = 0; a[1] = 1; //初始化第 1、2 个费波那契数
   for (n=2; n<N; n++) //循环，求第 3、4、 …、N 个费波那契数
      a[n] = a[n-2] + a[n-1];
   cout << "第" << N << "个费波那契数是：" << a[N-1] << endl;
}
```

例 2、从键盘输入 100 个整型数，把其中的最大者输出。

算法：

a）输入 100 个整型数，放入一个数组变量中。

b）先把第一个数作为当前的最大者。

c）从第二个数开始与当前的最大者进行比较，若小于或等于当前的最大者，则继续比较第三个、第四个、…；若大于当前的最大者,则把当前的数作为最大者，继续比较第三个、第四个、…

d）比较完第 100 个数后，输出当前的最大者。

程序

```
#include <iostream.h>
const int N=100;
int max(int x[],int num)
{ int i,j;
  j = 0; //j 用于记住当前最大者的下标，初始状态下把第 1 个数设
         //为当前最大者。
  for (i=1; i<num; i++) //循环，求最大者
    if (x[i] > x[j]) j = i; //若第 i+1 个数大于当前最大者，则
                            //把它设为当前最大者。
  return j;
}

void main()
{ int a[N],i,j;
  for (i=0; i<N; i++) cin >> a[i]; //输入 100 个整型数
  j = max(a,N); //调用函数 max 求数组 a 最大元素的下标
  cout << a[j] << endl; //输出最大者
}
```

例 3、从键盘输入 100 个整型数，把它们从小到大排序，然后输出排序后的结果。

算法：排序算法有很多种，下面介绍一种选择排序算法来实现。

从 100 个数中找出最大者，与第 100 个数交换位置；然后，从剩余的 99 个数中再找出最大者，与第 99 个数交换位置；…，一直到只有一个剩下的数结束。

```
#include <iostream.h>
const int N=100;
void main()
{ int a[N],i,j,temp;
  for (i=0; i<N; i++) cin >> a[i]; //输入 100 个整型数
  for (i=N; i>1; i--)
  { j = max(a,i);
```

```
            //交换 a[j]和 a[i-1]
            temp = a[j];
            a[j] = a[i-1];
            a[i-1] = temp;
        }
        for (i=0; i<100; i++) cout << a[i] << ' ';//输出排序结果
    }
```

5.2.2 二维数组

二维数组通常用于表示由固定多个具有类似矩阵元素次序的同类型数据所构成的复合数据。

1、二维数组类型的定义

(1)

```
int a[10][20];
```

(2)

```
typedef int A[20];
A a[10];
```

(3)

```
typedef int A[10][20];
A a;
```

2、二维数组元素的访问

```
int a[20][10];
a[0][0],a[0][1],…,a[0][9],
a[1][0],a[1][1],…,a[1][9],
…
a[19][0],a[19][1],…,a[19][9]
```

一般格式为：

<二维数组变量名>[<第 1 维下标>][<第 2 维下标>]

其中，<第 1 维下标>、<第 2 维下标>为整型表达式。

对二维数组的每一个元素访问一次通常需要一个二重循环：

```
for (int i=0; i<20; i++)
```

```
for (int j=0; j<10; j++)
{ …
  a[i][j] = ...
  …
}
```

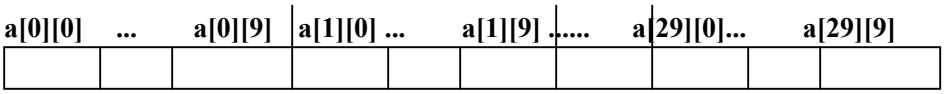
3、二维数组的初始化

```
int a[2][3]={{1,2,3},{4,5,6}};
int a[2][3]={{1,2},{4}};
int a[2][3]={1,2,3,4,5,6};
int a[2][3]={1,2,3,4};
int a[][3]={{1,2,3},{4,5,6},{7,8,9}};
```

4、二维数组的存贮

在内存中，二维数组变量将按行占用连续的存贮空间，所占空间的大小可由 sizeof 运算符计算。

```
int a[30][10];
```




```
cout << sizeof(a);
```

5、向函数传递二维数组

当需要把一个二维数组传给一个函数时，实参只需写数组变量名，函数形参为不带数组第一维大小的二维数组定义。

例：

```
void f(int x[][10],int num)
{    x[i][j]
}

void main()
{ int a[20][10],b[30][10],c[40][20];
   …
   f(a,20);
   …
   f(b,30);
   …
```

```
    f(c,40); //Error
    ...
}
```

6、二维数组降为一维数组处理

int a[20][10]；

a 可理解为一个拥有 20 个元素的一维数组，每个元素又是一个拥有 10 个元素的一维数组，即：

```
typedef int A[10];
A a[20];
```

例：求二维数组所表示的矩阵中每一行的最大元素。

```
…
int a[20][10];
…
int i,j;
for (i=0; i<20; i++)
{ j = max(a[i],10);
  cout << a[i][j] << endl;
}
```

求整个矩阵的最大元素值：

```
j = max(a[0],10*20);
cout << a[j/10][j%10];
```

7、应用

例、矩阵乘法

$A_{m,n} * B_{n,t} = C_{m,t}$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \quad (i=1..m, j=1..t)$$

```
int a[M][N],b[N][T],c[M][T];
int i,j,k;
for (i=0; i<M; i++) //对 c 的行循环
{ for (j=0; j<T; j++) //对 c 的列循环
  { c[i][j] = 0;
    for (k=0; k<N; k++) //计算 c[i][j]
```

```
            c[i][j] += a[i][k]*b[k][j];
        }
    }
```

5.3 结构(struct)与联合(union)

结构用于表示由固定多个不同类型的元素所构成的数据，如：一个学生的信息等。联合用于表示由固定多个不同类型的占用相同内存空间的元素所构成的数据。

1、结构类型的定义

格式：

(1) struct <结构类型名> {<成分定义表>};

(2) struct {<成分定义表>} <变量名表>;

其中，<成分定义表>为若干个成分变量定义。

例：

```
struct STUDENT
{ int no;
  char name[20];
  int age;
  enum {MALE, FEMALE} sex;
  char addr[80];
};
```

2、结构类型变量的定义

格式：

(1) struct STUDENT t1,t2,t3;

(2) STUDENT t1,t2,t3;

3、结构类型变量成分的访问

格式：

<结构类型变量>.<成分变量>

如：

```
t1.no = 1;
strcpy(t1.name,"张三");
```

```
t1.age = 18;
t1.sex = MALE;
…
```

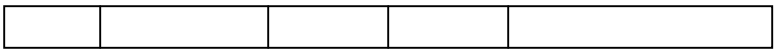
同类型的结构变量可以相互赋值，如：

```
t1 = t2;
```

4、结构类型变量的存贮

结构类型变量的各个元素依次占用一块连续的内存空间，结构变量占用的内存大小可以用 sizeof 计算。如：

```
STUDENT st;
```



5、向函数传递结构数据

当需要把一个结构数据传给一个函数时，实参写结构变量名，函数形参为结构类型变量定义，如：

```
void f(STUDENT s)
{ …s.no…
   …s.name…
}
void main()
{ STUDENT s1,s2;
   f(s1);
   f(s2);
}
```

上述的传递方式为值传递，效率不高，在介绍指针和引用类型时将介绍其它传递结构数据的方式。

6、例

从键盘输入 100 个学生的信息，然后把它们按学号从小到大输出。

```
#include <iostream.h>
const int STUDENT_NUM=100;
enum SEX {MALE, FEMALE};
struct STUDENT
```

```
{ int no;
   char name[20];
   int age;
   SEX sex;
   char addr[80];
   int scores[4];
} students[STUDENT_NUM];

void sort(STUDENT s[],int num)
{ int i,j;
   bool exchange;
   for (i=num; i>1; i--)
   { exchange = false;
      for (j=0; j<i-1; j++)
      { if (s[j+1].no < s[j].no)
        { STUDENT temp;
           temp = s[j];
           s[j] = s[j+1];
           s[j+1] = temp;
           exchange = true;
        }
      }
      if (!exchange) break;
   }
}

void main()
{ int i;
   char sex;

   for (i=0; i< STUDENT_NUM; i++) //输入 100 个学生信息
   { cin>>students[i].no>> students[i].name>> students[i].age;
      cin >> sex;
      if (sex == 'm')
        students[i].sex = MALE;
      else
        students[i].sex = FEMALE;
      cin >> students[i].addr;
   }
   sort(students, STUDENT_NUM);
```

```
    for (i=0; i< STUDENT_NUM; i++) //输出排序后的 100 个学生信息
    { cout << students[i].no << endl
           << students[i].name << endl
           << students[i].age << endl;
      if (students[i].sex == MALE)
        cout << "m" << endl;
      else
        cout << "f" << endl;
      cout << students[i].addr << endl;
      cout << "------------------------\n";
    }
  }
```

7、联合（共同体）

语法上，联合类型的定义与结构类型类似，不同之处在于把 struct 关键字换成 union，如：

```
union A
{ int i;
  char c;
  float f;
  double d;
  int j;
};
```

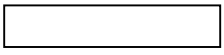
联合类型变量的定义、访问和向函数传递联合的形式同结构类型。

联合类型与结构类型的区别在于：联合类型的所有元素占用同一块内存空间，该内存空间大小为最大元素所需要的内存空间。

如：

```
A a;
```

a.i（a.c，a.f，a.d，a.j）



上述内存空间大小为 sizeof(double)

联合类型可以实现几个变量共享内存空间，在程序中不同的地方以不同的变量使用这块共享内存。如：

```
a.i = 12;
```

```
cout << a.j; //输出 12
cout << a.f; //输出什么？
```

联合类型还可以实现多态类型：一个类型有多种解释。

例：从键盘输入一组图形数据，保存在程序中的一个数组变量中，然后输出相应的图形。其中的图形可以是：线、矩形、圆。

```
#include <iostream.h>
enum FIGURE_TYPE { LINE, RECTANGLE, CIRCLE };

struct Line
{ FIGURE_TYPE t;
   double x1,y1,x2,y2;
};

struct Rectangle
{ FIGURE_TYPE t;
   double left,top,right,bottom;
};

struct Circle
{ FIGURE_TYPE t;
   double x,y,r;
};

union FIGURE
{ FIGURE_TYPE type;
   Line line;
   Rectangle rect;
   Circle circle;
};

FIGURE figures[100];

//输入图形信息
void get_figures()
{ int t;
   for (int i=0; i<100; i++)
   { cin >> t;
```

```
        switch (t)
        { case 0: //线
              figures[i].type = LINE;
              cin >> figures[i].line.x1
                  >> figures[i].line.y1
                  >> figures[i].line.x2
                  >> figures[i].line.y2;
              break;
          case 1: //矩形
              figures[i].type = RECTANGLE;
              cin >> figures[i].rect.left
                  >> figures[i].rect.top
                  >> figures[i].rect.right
                  >> figures[i].rect.bottom;
              break;
          case 2: //圆
              figures[i].type = CIRCLE;
              cin >> figures[i].circle.x
                  >> figures[i].circle.y
                  >> figures[i].circle.r;
              break;
        }
    }
}

//输出图形
void display_figures()
{ for (int i=0; i<100; i++)
    { switch (figures[i].type)
        { case LINE:
              draw_line(figures[i].line);
              break;
          case RECTANGLE:
              draw_rectangle(figures[i].rect);
              break;
          case CIRCLE:
              draw_circle(figures[i].circle);
              break;
        }
    }
```

```
}

void main()
{ get_figures();
   display_figures();
}
```

5.4 指针类型

5.4.1 指针的基本概念

1、指针类型的定义

<类型> *<指针变量>;

例如：

```
int *p;   //p 为一个指针变量
```

语义：

指针变量的值是另外一个变量的内存地址。如：上述指针变量 p 的值为某一个整型变量的内存地址。

当在一个定义（或声明）中定义（或声明）多个指针变量时，每个指针变量前都要有*，如：

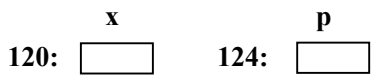
```
int *p,*q; //p 和 q 均为指针变量
int *p,q;   //p 为指针变量，q 为整型变量
```

也可以先用 typedef 定义一个指针类型，然后再用该指针类型定义指针变量，如：

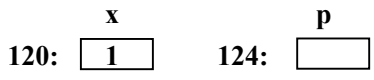
```
typedef int* Pointer;
Pointer p,q; // p 和 q 均为指针变量
```

2、取地址操作符(&)和间接访问操作符(*)

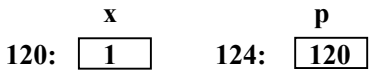
```
int x;
int *p;
```



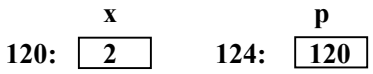

```
x = 1;
```



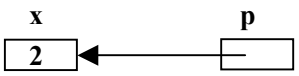
p = &x; // &x 表示取变量 x 的地址



***p = 2; // *p 表示取以 p 的值为地址的变量，*只能用于指针类型**



当一个变量的值为另一个变量的地址时，通常也说成：该变量指向另一个变量，如：指针变量 p 指向整型变量 x，可用下面的图示法表示：



因此，指针变量的值又称为指针，当一个指针变量没有指向任何其它变量时，可给它赋一个空指针：NULL(在 stdio.h 中声明)或 0，如：

p = NULL;

或

p = 0;

区别 * 的三种用法：

(1) 乘运算符

(2) 指针定义（或声明）符

(3) 取指针变量所指向的变量（间接访问）

3、指针变量的初始化：

```
int x;
int *p=&x;
```

特例：

char *p="ABCD"; //把字符串“ABCD”在内存中的首地址赋值给 p。

4、指针的运算

(1) 赋值

对于一个指针变量，可以把一个同类型的指针赋给它，如：

```
int x;
```

```
double y;
int *p,*q;
…
p = &x;   //OK，&x 的类型为：int*
p = &y;   //Error，&y 的类型为 double*
q = p;    //OK，q 指向 p 所指向的变量。
*q = *p; //把 p 所指向的变量的值赋给 q 所指向的变量。
p = NULL; //OK，使 p 不指向任何变量。
p = 120; //Error
p = (int *)120; //Ok
```

任何类型的指针可以赋给 void *类型的指针变量，如：

```
void *any_pointer;
any_pointer = &x;
any_pointer = &y;

… *any_pointer … //Error
… *((int *)any_pointer) … //OK
… *((double *)any_pointer) … //OK
```

(2) 加上或减去一个整型值

一个指针可以与一个整型值进行加或减运算，结果为同类型的指针，如：

```
int x;
int *p;
......
p = p + 1;
p = &x + 2;
p++;
p--;
```

当一个指针与一个整型值进行加（或减）运算时，实际加（或减）的值由该指针类型决定，如：

```
int *p;
double *q;
......
p++; //p 的值加 4（假设一个 int 型数据占 4 个字节内存空间）
q++; //q 的值加 8（假设一个 double 型数据占 8 个字节内存空间）
```

(3) 同类型的指针相减

同类型的指针可以相减，结果为整型值，如：

```
int *p,*q;
int offset;
offset = p - q;
```

相减的实际结果由指针类型确定，如：

```
int a[10];
p = &a[3];
q = &a[0];
```

p-q 的值为 3。

(4) 两个同类型的指针比较

两个同类型的指针可以进行比较运算：等于(==)、不等于(!=)、大于(>, >=)和小于(<, <=)。

(5) 指针的输出

```
int x=1;
int *p=&x;

cout << p;   //输出 p 的值(x 的地址)
cout << *p; //输出 x 的值
```

输出字符指针（char *）时，输出的不是指针值，而是该指针所指向的字符串，如：

```
char *p="ABCD";
…
cout << p;   //输出 p 指向的字符串，即:ABCD
cout << *p; //输出 p 指向的字符，即：A
cout << (void *)p //输出 p 的值，即字符'A'的地址
```

5.4.2 指针作为形参类型

1、向函数传递实参的地址

下面函数 swap 的目的是交换两个变量的值：

```
void swap(int x, int y)
```

```
{ int t;
   t = x;
   x = y;
   y = t;
}
void main()
{ int a=0,b=1;
   swap(a,b);
}
```

结果，a 和 b 的值没有进行交换！

C++的默认参数传递方式是值传递（数组除外），改变形参的值不会影响相应实参的值，这种参数传递方式能够保证所定义的函数不能通过修改形参的值来改变实参的值。

如果改变形参的值影响到实参，将带来函数的副作用问题，即：函数中改变了非局部量的值。函数的副作用将会使得程序的一些数学性质（如：运算的交换律）被破坏，例如：如果函数 f 改变了变量 a 的值，则：a+f(a) 和 f(a)+a 将得到不同的结果。

因此，函数应尽量不要有副作用。但是，在一些情况下，为了实现某些功能，需要函数的副作用，如上述的 swap 函数。

在 C++中，为了能在函数中改变实参的值，可以在参数传递时，把实参的地址传给函数，在函数中通过实参的地址间接地改变实参的值，这时，形参应定义成指针类型，如：

```
void f(int *p)
{ *p = 1;
}
void main()
{ int a=0;
   f(&a);
   cout << a; //结果为 1
}
```

利用指针类型，函数 swap 的正确实现可写成：

```
void swap(int *x, int *y)
{ int t;
   t = *x;
```

```
    *x = *y;
    *y = t;
}
void main()
{ int a=0,b=1;
   swap(&a,&b);
}
```

另外，向函数传递指针也能提高参数传递的效率，包括存贮效率和执行效率。C++中，数组参数的默认传递方式是把实参数组的首地址传给函数，以提高参数传递效率。

2、常量指针与指针常量

```
const int x=0;
int y,z;
const int *p; //p 为指向 int 型常量的指针（常量指针）
int *q;

p = &x;   //OK
q = &y;   //OK
*p = 1;   //Error
*q = 2;   //Ok

q = &x;   //Error!
p = &y;   //OK! Why?

int * const r=&y; //p 为指向 int 型变量的指针常量
r = &z; //Error
*r = 1; //Ok

const int * const p=&x; //表示什么？
```

在 C++中，指针作为一种参数传递机制，可以产生两个效果：

(1) 提高参数传递效率

(2) 通过形参来改变实参的值

在大部分情况下，程序中需要的是效果(1)，因为效果(2)会产生函数的副作用，使得程序的一些数学性质受到破坏。

要想使得指针只有(1)的效果而没有(2)的效果，除了由程序设计者来保证外，C++

语言也提供了措施，即常量指针（指向常量的指针）。

例：

```
void f(const int *p)
{ int t;
   .......
   t = *p; //Ok
   *p = 1; //Error
   …
}
void main()
{ int a;
   ......
   f(&a);
}
```

5.4.3 指针与数组

对数组元素的访问通常是通过下标来实现的。通过下标来访问数组元素时，首先计算下标的值，然后根据数组的首地址和下标的值，计算出欲访问的元素的内存地址。

频繁地采用下标（如：在循环中）访问数组元素的效率是不高的。通过指针来访问数组元素将会大大提高程序的执行效率。

1、数组元素的指针表示法

```
int a[10];
int *p;
int i;
```

p = &a[0]; ⇔ p = a; //一维数组的名表示数组第一个元素的地址，它是一个常量！
a[i] ⇔ *(a+i) ⇔ *(p+i) ⇔ p[i]
&a[i] ⇔ a+i ⇔ p+i ⇔ &p[i]

例：访问一维数组每个元素一次。

```
const int N=100;
int a[N];
int i,*p,*q;

for (i=0; i<N; i++)
{ … a[i] …     //计算地址：(char *)a+i*4
}
```

改进一：

```
p = a;
for (i=0; i<N; i++)
{ … *p …
   p++;
}
```

改进二：

```
p = a;
q = a + N-1;
do
{ … *p …
   p++;
} while (p != q);
```

对于一个 n（n>1）维数组，可以按一个一维数组理解，其元素的个数为第一维的大小，元素的类型为去掉第一维后的 n-1 维数组。

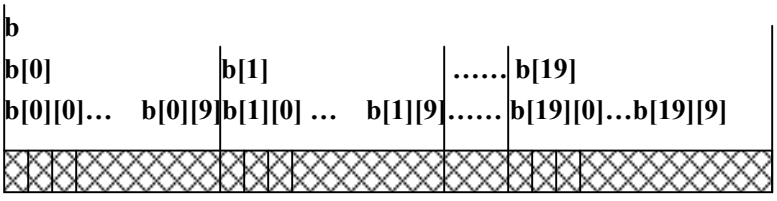
例：

```
int b[20][10];
```

可理解成：

```
typedef int A[10];
A b[20];
```

b 的存贮结构：




```
b                   // 为 20 个 A 类型的元素构成的一维数组变量,其元素为：
                    //    b[0],b[1],…,b[19]
b[i]            // 为  10 个 int 类型的元素构成的一维数组变量，其元素为：
                    //    b[i][0],b[i][1],…,b[i][9]
b[i][j]         // 为  int  类型的变量

b,&b[0]             // 为变量 b[0]的地址，类型为：A*
```

b+i,&b[i]　　// 为变量 b[i]的地址，类型为：A*
b[0],&b[0][0]　　// 为变量 b[0][0]的地址，类型为：int*
b[0]+i*10+j,&b[i][j]　　// 为变量 b[i][j]的地址，类型为：int*

int *q;
q = &b[0][0]; ⇔ q = b[0];

b[i][j] ⇔ *(&b[0][0]+i*10+j) ⇔ *(b[0]+i*10+j) ⇔ *(q+i*10+j) ⇔ q[i*10+j]

A *p; ⇔ int (*p)[10];
p = &b[0]; ⇔ p = b;

b[i][j] ⇔ *(*(&b[0]+i)+j) ⇔ *(*(b+i)+j) ⇔ *(*(p+i)+j) ⇔ p[i][j]

例、访问二维数组每个元素一次。

```
const int M=100;
const int N=200;
int a[M][N];
int i,j;

for (i=0; i<M; i++)
   for (j=0; j<N; j++)
   { … a[i][j] … //计算地址：(char *)&a[0][0]+(i*N+j)*4
   }
```

改进：

```
int *p;
p = &a[0][0];
for (i=0; i<M*N; i++)
{ … *p …
p++;
}
```

2、数组形参的指针表示法

```
void f(int x[],int num); ⇔ void f(int *x, int num);
void g(int y[][10], int num); ⇔ void g(int (*y)[10], int num);
void main()
{ int a[10];
```

```
    int b[20][10];
    f(a,10); ⇔ f(&a[0],10);
    g(b,20); ⇔ g(&b[0],20);
    f(b[0],10);
}
```

3、函数 main 的参数

在 C++中，主函数 main 可以带有形式参数，用于接收由操作系统传给它的数据。如果在定义 main 时没有给出形参定义，表明程序不需要操作系统提供数据。main 也可以有返回值，用于告诉操作系统程序结束的原因（正常或异常结束）。

带参数和返回值的函数 main 的原型为：

```
int main(int argc, char *argv[], char *env[]);
```

其中，**argc** 表示参数的个数，**argv** 表示命令行参数，**env** 表示环境参数。

例如，对于下面的 **dos** 命令行：

```
c>copy file1 file2
```

操作系统将启动 copy 程序，调用其 main 函数，把 3 传给 argc,把字符串"copy"、"file1"、"file2"的地址传给 argv，把各个环境字符串的地址传给 env。

5.4.4 指针与结构

1、指向结构的指针

```
struct A
{ int i;
  double d;
  char ch;
};

A a;
A *p;

p = &a;
```

2、通过指针访问结构成分

(*p).i ⇔ p->i

3、通过指针向函数传递结构

通过值传递方式向函数传递结构数据的效率是不高的，为了提高效率，可把结构数据的地址传给函数，在函数中通过间接方式（指针）访问结构数据，如：

```
void f(A *p)
{ …
   … (*p).i  或 p->i …
   … (*p).d 或 p->d …
   … (*p).ch 或 p->ch …
}
void main()
{ A a;
   f(&a);
}
```

5.4.5 动态变量

程序中往往需要用到这样一些变量：变量的大小或个数要到程序执行的时候才能确定，如元素个数可变的数组等。

在程序设计中采用动态变量设施来解决上述问题。所谓动态变量是指：从静态的程序中无法确定它们的存在，只有当程序运行起来，随着程序的运行，它们动态产生、消亡。虽然，局部变量也是动态产生、消亡，但在程序运行前，编译程序已经知道它们的存在，因此，局部变量不是上述意义下的动态变量。

注意：不要把这里的动态变量与 C++中的静态(static)变量参照起来理解，因为它们属于不同的范畴，动态变量是程序设计中的概念（与语言无关），而 static 是 C++语言中的概念，它用来声明变量的作用域或生存期。

动态变量的内存空间分配在程序的堆区。对动态变量的访问是通过指向动态变量的指针变量来实现的。

1、动态变量的创建

C++提供了两种创建动态变量的设施：操作符 new 和库函数 malloc，它们的格式为：

(1) new <类型名>

产生一个类型由<类型名>指定类型的动态变量，结果为该动态变量的地址。

例：

```
int *p;
```

p = new int; //产生一个动态的整型变量，p 指向该变量。

***p = 1; //通过指针变量来访问动态变量。**

(2) new <类型名>[<整型表达式 1>]...[<整型表达式 n>]

产生一个动态的 n 维数组，数组元素的类型由<类型名>表示。结果为第一维的第一个元素的地址。

例：

int *p;

p = new int[20]; //产生一个由 20 个整型元素所构成的一维动态数组。

int (*q)[20];

q = new int[10][20]; //产生一个由 10x20 个整型元素所构成的二维动态数组。

(3) void *malloc(unsigned int size);

在程序的堆区中分配一块大小为 size 的内存空间，返回该内存空间的首地址，如果该空间用于某个具体类型的动态变量，则需对返回值类型进行强制类型转换。malloc 在 malloc.h 中声明。

例：

int *p,*q;

p = (int *)malloc(sizeof(int)); //p 指向一个整型动态变量。

q = (int *)malloc(sizeof(int)*20); // p 指向一个整型一维动态数组变量。

new 与 malloc 的主要区别在于：

a）new 自动返回相应类型的指针，malloc 要作强制类型转换。

b）如果创建的是动态对象，new 会去调用相应类的构造函数，malloc 则否。

2、动态变量的撤消

动态变量不会自动消亡，在程序运行期间，如果不再需要某个动态变量，应显式地使之消亡，否则这些动态变量将造成内存空间浪费。

要撤消（使之消亡）某个动态变量，可以用操作符 delete 或库函数 free 来实现。一般情况下，用 new 产生的动态变量，用 delete 使之消亡；用 malloc 产生的动态变量，用 free 使之消亡。

delete 与 free 的格式如下：

(1) delete <指针变量>

撤消<指针变量>所指向的动态变量。

例：

```
int *p=new int;
...
delete p;
```

(2) delete []<指针变量>

撤消<指针变量>所指向的动态数组变量，

```
int *p=new int[20];
...
delete []p;
```

(3) void free(void *p)

释放 p 所指向的内存空间。

例：

```
int *p=(int *)malloc(sizeof(int));
int *q=(int *)malloc(sizeof(int)*20);
...
free(p);
free(q);
```

注意：

不能用 delete 和 free 撤消非动态变量，否则产生系统错误。

3、动态变量的应用－－链表

通常用数组来表示由多个同类型的具有顺序关系的元素所构成的复合数据。如果在程序运行前就能确定数组元素的个数，如：对输入的 100 个数进行排序，则程序中可采用非动态数组来存贮这些这些数据，这时，数组变量可定义为：

```
int a[100];
for (int i=0; i<100; i++) cin >> a[i];
sort(a,100);
```

如果程序运行前无法确定数组元素的个数，但在程序运行中访问数组元素前能够知道数组元素的个数，如：对输入的若干个数进行排序，输入时先输入数的个数，然后再

输入各个数，则可采用动态数组来存贮它们：

```
int n;
int *p;
cin >> n;
p = new int[n];
for (int i=0; i<n; i++) cin >> p[i];
sort(p,n);
```

如果程序运行前不知道数组元素的个数，程序运行中访问数组元素前也无法确定数组元素的个数，如：对输入的若干个数进行排序，输入时先输入各个数，然后再输入一个结束标记（如：-1），这时，可以按以下方式实现：

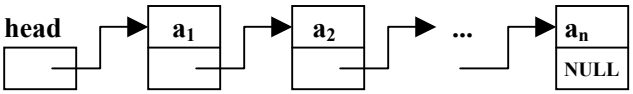
```
int len=20;
int i=0,n;
int *p=new int[len];
cin >> n;
while (n != -1)
{ if (i < len)
   { p[i] = n;
     i++;
   }
   else
   { int *q=new int[len+20];
     len += 20;
     for (int j=0; j<i; j++) q[j] = p[j];
     delete p;
     p = q;
     p[i] = n;
     i++;
   }
   cin >> n;
}
sort(p,i);
```

用数组来表示一组具有顺序关系的元素所构成的数据时，除了要考虑数组对元素个数的限制外，当在数组中增加或删除元素时，将会面临数组元素的大量移动。采用链表结构可以避免数组的一些问题。

链表是这样的一种数据（结构）：它由若干个（个数不定）同类型的元素构成的线性结构；链表中的每一个元素除了本身的值以外，还包含一个（或多个）指针，指向链表中的其它元素。如果每个元素只包含一个指针，则称为单链表，否则称为多链表。上述的定义隐含着链表元素在内存中不必存放在连续的空间内。实际上，链表不是一种抽象结构，而是一种存贮结构。

下面以单链表为例，介绍链表的基本内容。

下图给出了一个单链表的图示：



该单链表由若干结点（元素）构成，每个结点除了具有一个存贮结点值的域以外，还包含一个指针域，指向下一个结点，最后一个结点的指针为空（NULL）。另外，该单链表还有一个头指针（head），指向它的第一个结点。

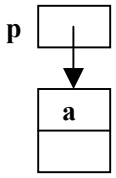
(1) 结点类型和表头指针变量的定义

```
struct NODE
{ int content;
   NODE *next;
}; //结点的类型定义
NODE *head=NULL; //头指针变量定义
```

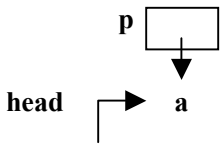
(2) 结点的插入

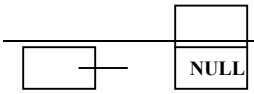
a）生成新结点：

```
NODE *p=new NODE;
cin >> p->content;
```



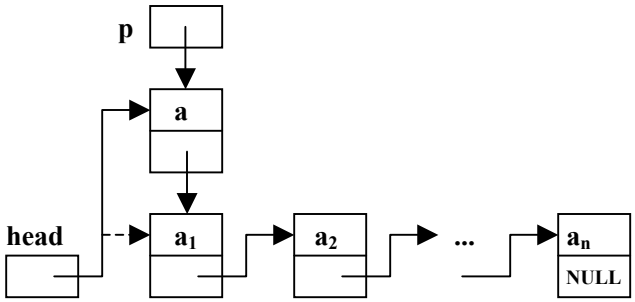
b）如果链表为空（创建第一个结点时），则：





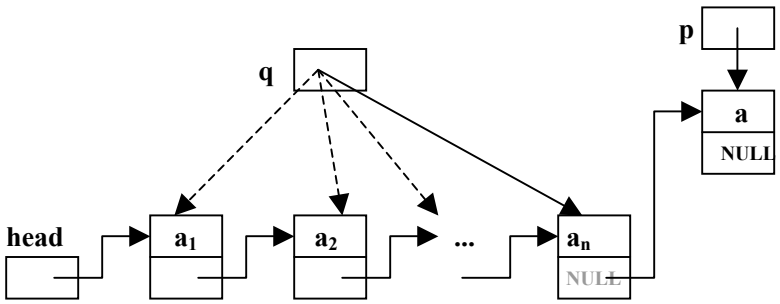
head = p;
p->next = NULL;(或，head->next = NULL;)

c）如果新结点插在表头，则：



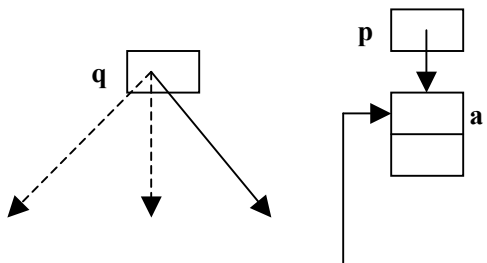
p->next = head;
head = p;

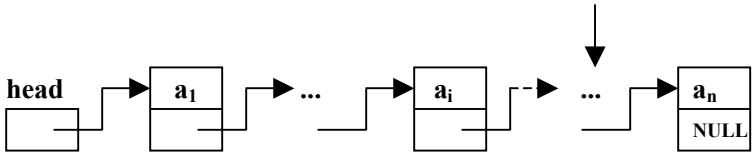
d）如果插在表尾，则：



NODE *q=head;
while (q->next != NULL) q = q->next;
q->next = p;
p->next = NULL;

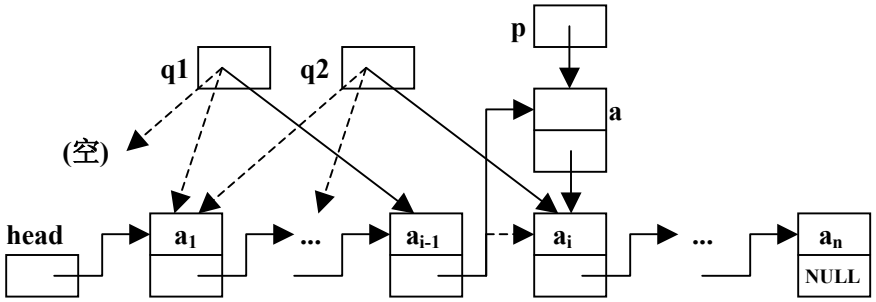
e）如果插在链表中某一个结点（a_i）的后面，则：






```
NODE *q=head;
while (q != NULL && q->content != ai) q = q->next;
if (q != NULL) //存在 ai
{ p->next = q->next;
   q->next = p;
}
else //  不存在 ai
   cout << "Not found!";
```

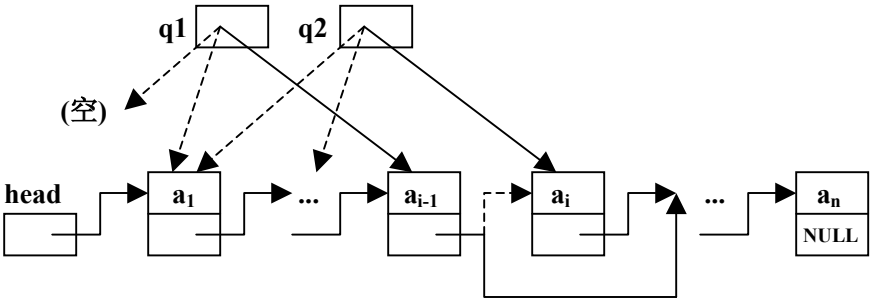
f）如果插在链表中某一个结点（a_i）的前面，则：




```
NODE *q1=NULL,*q2=head;
while (q2 != NULL && q2->content != ai)
{ q1 = q2;
   q2 = q2->next;
}
if (q2 != NULL) //存在 ai
{ p->next = q2;
   if (q1 != NULL) // ai 不是第一个结点  head != q2
      q1->next = p;
   else // ai 是第一个结点
      head = p;
}
else //不存在 ai
   cout << "Not found!";
```

(3) 结点的删除

假设要删除的结点的值为 a_i：




```
NODE *q1=NULL,*q2=head;
while (q2 != NULL && q2->content != ai)
{ q1 = q2;
   q2 = q2->next;
}
if (q2 != NULL) //存在 ai
{ if (q1 != NULL) // ai 不是第一个结点
      q1->next = q2->next;
   else // ai 是第一个结点
      head = q2->next; //或：head = head->next;
   delete q2;
}
else //不存在 ai
   cout << "Not found!";
```

5.4.6 函数指针

1、指向函数的指针

C++中可以定义一个指针变量，使其指向一个函数，如：

```
double f(int x)
{ …
}
int g()
{
}
void main()
{ double (*fp)(int);
```

```
    fp = f;   //⇔ fp = &f;

    fp(10);   // ⇔   (*fp)(10); ⇔   f(10);

    fp = g; //Error，类型不一致
}
```

指向函数的指针类型也可以用 typedef 给出，格式如下：

typedef <类型>(*<函数指针类型名>)(<参数表>);

如：

```
typedef double (*FP)(int);
FP fp;
```

2、向函数传递函数

C++允许把一个函数作为一个参数传给另一个函数，如：

```
int f(int);
int g(int);
int func(int (*fp)(int x))
{ int i;
   …
   (*fp)(i+10);
}
void main()
{ …
   func(&f);
   func(&g);
}
```

例：编写一个函数，使其能计算任意一个一元函数在一个区间上的定积分。

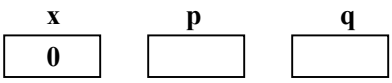
```
#include <math.h>
double integrate(double (*f)(double),double a, double b)
{ … (*f)(x),a,b, …

}
double my_func(double x)
```

```
{ …
}
void main()
{ …
   integrate(sin,0,1);
   integrate(cos,1,2);
   integrate(my_func,1,10);
   …
}
```

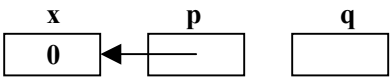
5.4.7 多级指针

在定义指针变量时，指针变量所指向的变量的类型可以是任意类型。如果一个指针变量所指向的变量的类型为指针类型，则为多级指针，如：

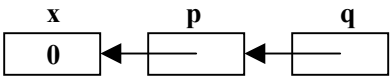
```
int x=0;
int *p;
int **q;   //q 为指针变量，它将指向另一个指针变量，这个指针变量指向一个整型变量。
```



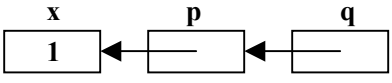

```
p = &x;   //p 指向 x
```



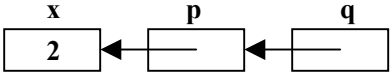

```
q = &p;   //q 指向 p
```



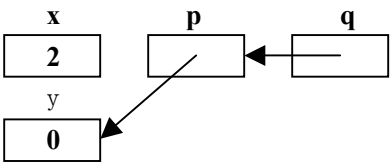

```
q = &x;   //Error

*p = 1;   //x 的值变为 1
```




```
**q = 2; //x 的值变为 2
```



```
int y=0;
*q = &y; //p 指向 y
```



注意：

如果一个指针变量没有初始化或赋值，访问它所指向的变量将会导致运行时刻的严重错误，如：

```
int x;
int *p;
int **q;
*p = 1; //Error
*q = &x; //Error
**q = 2; //Error
```

例 1、编写一个函数，交换两个指针变量的值。

```
void swap(int **x, int **y)
{ int *t;
   t = *x;
   *x = *y;
   *y = t;
}
void main()
{ int a=0,b=1;
   int *p=&a,*q=&b;
   swap(&p,&q);
}
```

例 2、编写一个函数，在一个整型数组中查找某个指定的元素，若找到，则函数返回真，并把该数组元素的地址从一个参数中返回给调用者；否则，函数返回假。

```
bool find(int key,int x[],int num,int *p)
{ int i=0;
   while (i < num)
   { if (x[i] == key)
      { p = &x[i];
         return true;
```

```
        }
        i++;
    }
    return false;
}
void main()
{ int a[20],*q;
    ......
    q = a;
    if (find(12,a,20,q) == true)
        cout << q << endl;
    else
        cout << "not found" << endl;
}
```

上述函数 find 是否正确？

正确的 find 的写法是：

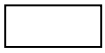
```
bool find(int key,int x[],int num,int **p)
{ int i=0;
    while (i < num)
    { if (x[i] == key)
        { *p = &x[i];
            return true;
        }
        i++;
    }
    return false;
}
void main()
{ int a[20],*q;
    ......
    if (find(12,a,20,&q) == true)
        cout << q << endl;
    else
        cout << "not found" << endl;
}
```

5.5 引用类型

1、定义

C++提供了引用类型，通过引用类型可以定义一个变量，它与另一个变量占用相同的内存空间，或为一个变量取一个别名。

```
int x=0;
int &y=x;   //y 引用 x，y 与 x 占用相同的内存空间
```




```
x = 1;
cout << y; //结果为：1

y = 2;
cout << x; //结果为：2
```

注意：

(1) 引用变量和被引用变量应具有相同的类型

(2) 引用变量定义中的&不是取地址操作符

(3) 定义引用变量时必须要有初始化

2、引用类型的应用

引用类型主要用于函数参数传递：引用类型形参与相应的实参占用相同的内存空间，改变引用类型形参的值，相应实参的值也随着变化。

例：写一个函数交换两个变量的值。

```
void swap(int &x, int &y)
{ int t;
   t = x;
   x = y;
   y = t;
}
void main()
{ int a=0,b=1;
   swap(a,b);
   cout << a << ',' << b; //结果为：1 ,  0
}
```

如果不允许通过引用形参改变相应实参的值，则定义引用形参时可加上 const 关键词。

```
void f(const int& x)
{ ......
   x = 1; //Error
   ......
}
```

3、引用类型与指针类型的区别

引用类型和指针类型都可以实现通过一个变量访问另一个变量，但访问的形式不同：引用是直接访问，而指针是间接访问。

引用变量没有自己单独的内存空间，它与被引用的变量共享内存空间；指针变量有自己的内存空间，它独立于所指向的变量所具有的内存空间。引用变量不能再引用其它变量；而指针变量可以指向其它同类型的变量。

在作为函数参数时，引用类型参数的实参是一个变量，而指针类型参数的实参是一个变量的地址。从实现的角度讲，引用类型和指针类型参数传递是一样的。

4、函数返回值类型为指针或引用

函数返回值的类型可以是引用或指针类型。

例：

```
int max1(int x[], int num)
{ int m,i;
   m = x[0];
   for (i=1; i<num; i++)
      if (x[i] > m) m = x[i];
   return m;
}

int *max2(int x[], int num)
{ int *p,*q;
   p = x;
   q = x+1;
   while (num > 1)
   { if (*q > *p) p = q;
     q++;
     num--;
   }
   return p;
}
```

```
int &max3(int x[], int num)
{ int i,j;
  j = 0;
  for (i=1; i<num; i++)
     if (x[i] > x[j]) j = i;
  return x[j];
}

void main()
{ int a[100];
  .......
  cout << max1(a,100) << endl;
  cout << *max2(a,100) << endl;
  cout << max3(a,100) << endl;

  max1(a,100) = 1; //Error
  *max2(a,100) = 1; //Ok
  max3(a,100) = 1; //Ok
}
```

一般来说，如果函数返回值的类型是引用或指针类型，则函数不应该把局部量或局部量的地址作为返回值返回。

例：

```
int f1()
{ int x;
    x = 1;
    return x;
}

int &f2()
{ int y;
    y = 0;
    return y;
}

int *f3()
{ int z;
  z = 2;
  return &z;
```

}

对于上述的三个函数的调用，将会出现下面的现象：

int *p;
p = f3();
***p + f1() 的值为 2，而不是 3, Why?**

f1()+f2() 的值为 1，而
f2()+f1() 的值为 2，Why?

第6章 数据抽象－－类

6.1 从面向过程到面向对象

6.1.1 什么是面向对象程序设计

1、例子：实现整型“栈”数据类型。

栈是一种由若干个按线性次序排列的元素所构成的复合数据，对栈只能实施两种操作：进栈（增加一个元素）和退栈（删除一个元素），并且这两个操作必须在栈的同一端（栈顶）进行。后进先出（Last In First Out,简称 LIFO）是栈的一个重要性质。

(1) 非面向对象方案

```
#include <iostream.h>
//定义栈数据类型
#define STACK_SIZE 100
struct Stack
{ int top;
   int buffer[STACK_SIZE];
};
void init(Stack &s)
{ s.top = -1;
}
bool push(Stack &s, int i);
{ if (s.top == STACK_SIZE-1)
   { cout << “Stack is overflow.\n”;
     return false;
   }
   else
   { s.top++;
      s.buffer[s.top] = i;
      return true;
   }
}
bool pop(Stack &s, int &i)
{ if (s.top == -1)
   { cout <<“Stack is empty.\n”;
      return false;
```

```
    }
    else
    { i = s.buffer[s.top];
      s.top--;
      return true;
    }
  }

  ……
  //使用栈类型数据
  Stack st;
  int x;
  init(st);
  push(st,12);
  pop(st,x);
```

或，

```
  Stack st;
  int x;
  st.top = -1;
  st.top++;
  st.buffer[st.top] = 12;
  x = st.buffer[st.top];
  st.top--;
```

(2) 面向对象方案

```
  #include <iostream.h>
  //定义栈数据类型
  #define STACK_SIZE 100
  class Stack
  {   int top;
      int buffer[STACK_SIZE];
    public:
      Stack() { top = -1; }
      bool push(int i);
      { if (top == STACK_SIZE-1)
        { cout << "Stack is overflow.\n";
          return false;
        }
        else
```

```
        { top++;
          buffer[top] = i;
          return true;
        }
    }
    bool pop(int &i);
    { if (top == -1)
        { cout << "Stack is empty.\n";
          return false;
        }
        else
        { i = buffer[top];
          top--;
          return true;
        }
    }
};
……
//使用栈类型数据
Stack st;
int x;
st.push(12);
st.pop(x);
st.top = -1; //error
st.top++; //error
st.buffer[st.top] = 12; //error
```

(3) 两种方案的简单比较

a) 方案(1)需要显式地对栈进行初始化，方案(2)则否。

b) 方案(1)中对栈的操作可以通过提供的函数来实现，也可以直接在栈的数据表示上进行；方案(2)中只能通过提供的函数来操作栈。

c) 当栈的实现发生变化（如把数组改成了链表）时，方案(1)往往要影响到栈的使用者，方案(2)则否。

2、面向对象程序设计的定义

面向对象程序设计是把程序构造成由若干对象组成，每个对象由一些数据以及对这些数据所实施的操作构成；对数据的操作是通过向包含数据的对象发送消息来实现(调用对象的操作)；对象的属性(数据与操作)由(对象)类来描述，一个类所描述的属性可

以从其它的类继承。

6.1.2 为什么要面向对象

一个好的软件开发方法或技术的评价标准：开发效率和软件质量保证。开发效率指方法使用的难易程度和方法缩短开发周期的程度等。软件质量包括外部质量和内部质量。

软件的外部质量是指软件中与用户有关的质量因素，这里的用户包括最终用户和二次开发用户。包括：正确性(Correctness)、效率(Efficiency)、健壮性或鲁棒性(Robustness)、可靠性(Reliability)、可用性(Usability)和可复用性(Reusability)等方面。

软件的内部质量是指软件中与软件开发人员有关的质量因素。包括：可读性(Readability)和可维护性(maintainability)等。

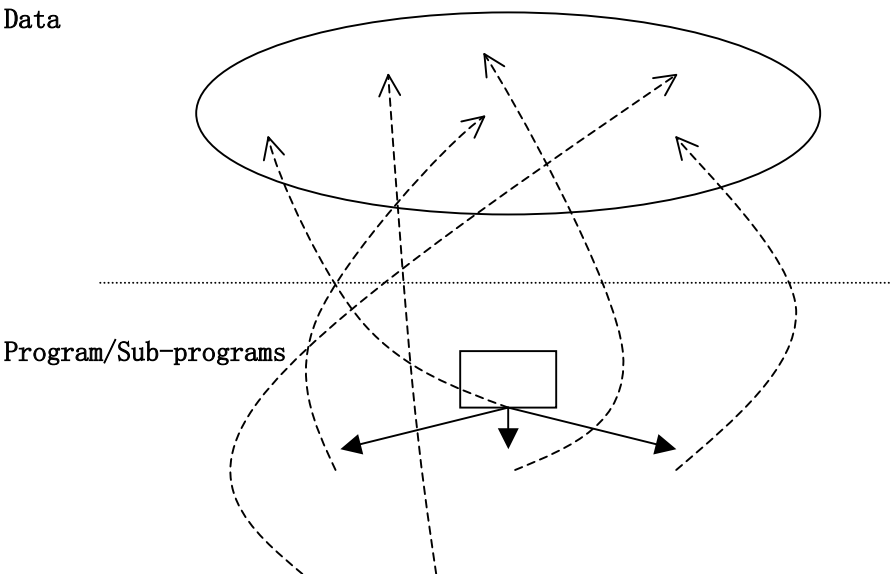
下面就能够提高软件开发效率和保证软件质量的几个程序特性来说明面向对象程序设计的优势：

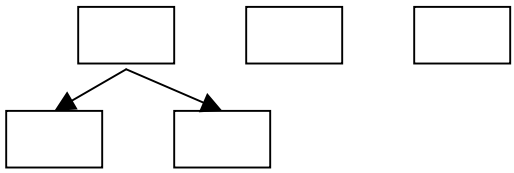
1、抽象(Abstraction)

处理大而复杂问题的重要手段是抽象：强调事物本质的东西。对程序抽象而言，一个语言结构的抽象强调的是该结构外部可观察到的行为，与该结构的内部实现无关。

(1) 过程抽象(Procedural Abstraction)

过程抽象是指把程序的一些功能抽象为子程序（过程），它把子程序的接口和实现分开，使用者只需要知道子程序的接口(功能和参数)而不需要关心其内部实现，适合于基于功能分解的逐步精化(Step-wise)程序设计。其不足之处在于：数据与操作的描述分离，缺乏数据保护；程序难以复用；不能适应功能的变化。



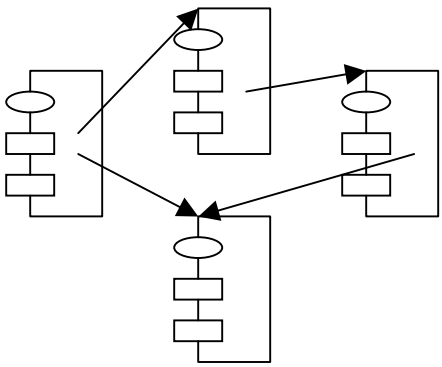


过程式程序设计支持的是过程抽象。

(2) 数据抽象(Data Abstraction)

数据抽象是指以数据为中心，把数据及其操作作为一个整体（对象）来进行描述，对数据的操作由包含数据的对象来提供。数据抽象一方面加强了数据保护；另一方面它是对待解决问题的自然、真实的模拟，软件能够适应需求的变化，并且有利于软件的复用。

Objects



面向对象程序设计支持的是数据抽象。

2、封装(Encapsulation)

封装是指把一个语言结构的具体实现细节作为一个黑匣子对该结构的使用者隐藏起来的一种机制，从而符合信息隐藏(Information Hiding)原则，有利于保证软件的正确性和提高软件的可复用性以及可维护性。

过多地暴露实现细节，无论是对使用者，还是对实现者都是不利的。对于使用者而言，如果其功能的正常执行要依赖所使用的语言结构的内部实现，那么，当所使用的语言结构的内部实现发生变化时，其必须也要作相应改变；对实现者而言，如果过多地暴露实现细节，则不得不谨慎地处理任何实现上的改变，从而不至于影响太多的使用者。

封装考虑的是内部实现，抽象考虑的是外部行为。

(1) 过程封装

过程封装实现了操作的封装，而数据是公开的，缺乏对数据保护。过程式程序设计支持的是过程封装。

(2) 数据封装

数据封装实现了数据及其操作的封装，加强了数据保护。面向对象程序设计支持的是数据封装。

3、模块化(Modularity)

模块化是处理大而复杂问题的重要手段，同时也是保证软件质量的有力措施，一个好的软件开发方法应能支持模块化。

逻辑上讲，模块是指程序中相关的结构所构成的一个相对独立的单位，模块化是指根据某些原则把程序分成若干模块。物理上讲，模块是指可以分别编译的程序单位。模块化的好处是：程序容易设计、理解与维护。

一个模块包含接口和实现两部分。接口起到了模块的设计者和使用者之间的一种约束作用；实现给出了满足接口约束的模块内部的具体实现。

划分模块的一般标准是：内聚性最大和耦合度最小原则。具体体现为模块的可分解性、可结合性、可理解性、连续性和保护性。

过程式程序设计主要以功能来划分模块，具体如何划分，自由度较大，模块的边界比较模糊。并且，一旦功能发生变化，有可能引起模块的重新划分。

面向对象程序设计对模块有更好的支持，它以对象/类作为模块划分的依据，模块边界比较清晰，并且划分出的模块结构比较稳定。

4、软件复用(Reuse)

软件复用的层次：代码复用、设计过程复用和分析方案复用。代码复用最直接、最广泛。

(1) 传统的面向过程的复用机制：源代码的剪裁和子程序库。

(2) 面向对象的复用机制：继承、聚集和类库。

5、软件开发过程

软件从无到有、直到消亡，往往要经历：软件需求分析、软件设计、实现（编程）、测试、运行以及维护等阶段，这个过程称为软件生命周期。

由于功能是易变的，采用基于自顶向下、功能分解的软件开发方法（如：结构化方法）所面临的问题是：开发出的软件系统难以适应需求（功能）的变化。

采用面向对象方法开发软件能够：

(1) 减小各阶段之间的语义间隙，使得开发过程平稳过渡。

(2) 提高软件的可维护性，特别是对需求变化的适应性。

6.1.3 面向对象程序设计基本内容

1、对象与类(Object&Class)

对象是由数据（数据成员、成员变量、实例变量、对象的局部变量）及其操作（方法，成员函数、消息处理过程）所构成的封装体。

类是对象特性的描述，一个类刻画了一组具有相同特性的对象，是创建对象的模板。对象是类的实例。

对象属于值的范畴，而类属于类型的范畴。

对象与类实现数据抽象、封装、模块

2、继承(Inheritance)

继承是实现软件复用的一种重要设施。

在继承机制中存在两种类：父类（基类）与子类（派生类），子类复用父类的一些特性。子类除了包含父类的属性以外，也可以定义新的属性，或重新定义父类的属性。

继承可分为：单继承（一个类最多有一个直接父类）与多继承（一个类可以有多个直接父类）。

3、多态(Polymorphism)

多态的一般含义是：某一论域中的一个元素可以有多种解释。具体到程序语言，则有以下两个含义：

(1) 相同的语言结构可以代表不同类型的实体；

一名多用或重载（Overloading），如：操作符重载与函数重载。

(2) 相同的语言结构可以对不同类型的实体进行操作。

类属（Genericity），如：类属函数与类属类。

面向对象程序设计除了支持上述的多态外，还支持一种独特的多态：一个公共的消息集可以发送到不同种类的对象，从而会得到不同的处理。这种多态的出现是由对象类之间存在继承关系所产生的。

多态带来的好处是：提高程序设计的灵活性、实现高层软件的复用。

与多态相关的一个机制是绑定（联编、定联）(Binding)，它是指确定对多态元素使用的过程，即确定某个使用点使用的是多态元素的那一种形式。绑定分为：静态（前期，Early Binding）与动态（后期，Late Binding）绑定。静态绑定是指在编译时刻确定；动态绑定是指在运行时刻实现。大多数多态可采用静态绑定，有些多态（如：消息的多态）往往要动态绑定。

6.2 类的定义

类定义的格式如下：

```
class <类名>
{ <成员定义或声明表>
};
```

其中的成员可以是数据成员（常量和变量），也可以是成员函数。

对于数据成员，在类定义中给出的是声明。对于成员函数，在类定义中给出的是定义或声明，如果类定义中只给出了成员函数的声明，则在类定义外给出成员函数的定义。

另外，在类定义中还给出类成员的访问控制描述。

例：

```
class CDate //类定义
{ public: //访问控制声明
      void SetDate(int y, int m, int d);  //成员函数声明
      int   IsLeapYear();  //成员函数声明
      void Print() { cout << year << ":" << month << ":" << day << endl; } //成员函数定义
   private: //访问控制声明
      int year,month,day; //数据成员声明
};
void CDate::SetDate(int y, int m, int d) //成员函数定义
{ .......
}
int   CDate::IsLeapYear()//成员函数定义
{ ......
```

```
}
```

在上述的类定义中，除了成员的声明和定义外，还用 public 和 private 给出了成员的访问控制。

6.2.1 数据成员

类的数据成员可以是常量和变量，其中的变量又称成员变量。数据成员的声明格式与非成员数据声明的格式相同，不过，需要注意的是：

1、类定义中给出的是数据成员的声明，不能对它们赋初值。

2、数据成员的类型可以是任意类型（类），但是，在声明一个数据成员的类型时，如果未见到相应的类型定义或相应的类型未定义完，则该数据成员只能声明成指针或引用类型。

```
class A;
class B
{ A a; //Error
  B b; //Error
  A *p; //OK
  B *q; //OK
  A& aa; //OK
  B& bb; //OK
}
```

数据成员的初始化应放在构造函数的成员初始化表或构造函数的函数体中描述（详见 6.3.1）。

6.2.2 成员函数

成员函数的定义可以放在类定义中，也可以放在类定义外。

1、在类定义中定义成员函数。

```
class CDate
{ public:
    void SetDate(int y, int m, int d)
    { year = y;
      month = m;
      day = d;
    }
    int   IsLeapYear()
    { return (year%4 == 0 && year%100 != 0) || (year%400==0);
```

```
    }
    void Print()
    { cout<<year<<"."<<month<<"."<<day<<endl;
    }
  private:
    int year,month,day;
};
```

如果成员函数的定义放在类定义中，则表示建议编译程序按内联函数处理之。

2、在类定义外定义成员函数：

如果在类定义外定义成员函数，则在类定义中应给出成员函数的声明，并且在类定义外定义成员函数时，应在返回类型和函数名之间加上:

<类名>::

如：

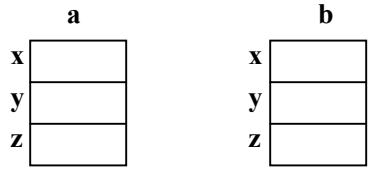
```
class CDate
{ public:
    void SetDate(int y, int m, int d);
    int   IsLeapYear();
    void Print();
  private:
    int year,month,day;
};
void CDate::SetDate(int y, int m, int d)
{ year = y;
  month = m;
  day = d;
}
int   CDate::IsLeapYear()
{ return (year%4 == 0 && year%100 != 0) || (year%400==0);
}
void CDate::Print()
{ cout<<year<<"."<<month<<"."<<day<<endl;
}
```

一般情况下，类定义放在头文件（.h）中，类外定义的成员函数放在实现文件（.cpp）中。

6.2.3 类成员的访问：对象

类与对象是不可分割的两个不同概念。类属于类型范畴，用于描述对象的属性。对象属于值的范畴，是类的实例。类中描述的成员应该属于实例化后的对象。类定义中声明的数据成员（除静态数据成员外），对该类的每个对象都有一个拷贝。如：

```
class A
{ public:
      f();
      g();
   private:
      int x,y,z;
};
......
A a,b;
```



类定义中的成员函数，对该类的所有对象只有一个拷贝。当对一个类的某对象调用该类的成员函数时，相应对象的地址将作为一个隐含的参数传给成员函数（详见下面的2）。

类是一个静态的概念，它存在于静态程序（程序运行前）。对象是一个动态的概念，它存在于动态程序（程序运行中）。

1、在非成员函数中访问类成员

在一个类的非成员函数中访问该类的成员（数据成员和成员函数），需通过该类的对象来进行，如：

```
class A
{ public:
    int x;
    void f();
};
void q()
{ A a; //创建 A 类的一个局部对象 a。
  a.x = 1;
  a.f();
```

```
    A *p=new A; //创建 A 类的一个动态对象，p 指向之。
    p->x = 2;
    p->f();
}
```

通过对象来访问类的成员时要受到类成员访问控制的限制，如：

```
class A
{ private:
     int x;
     void f();
};
......
q()
{ A a;
   a.x = 1; //Error
   a.f(); //Error
   A *p=new A;
   p->x = 2; //Error
   p->f(); //Error
}
```

类成员的访问控制将在 6.2.4 中进行介绍。

2、在成员函数中访问类成员

在一个类的成员函数中访问该类的成员（数据成员和成员函数），直接访问即可，如：

```
class A
{ public:
     void f();
     void q(int i) { x = i; f(); }
  private:
     int x;
};
......
A a,b;
a.q(1); //在 A::q 中访问的是 a.x
b.q(2); //在 A::q 中访问的是 b.x
```

问题：

对于下面的非成员函数 func：

```
void func(A *p)
{ …
  p->q(1);
  …
}
```

如果在 A::f 中调用 func，即：

```
class A
{ public:
    int x;
    void f() { func(...); }
    void q(int i) { x = i; f(); }
};
```

对于：

```
A a,b;
```

要求：

(1) 当调用 a.f()时，在 A::f 中调用 func(&a)

(2) 当调用 b.f()时，在 A::f 中调用 func(&b)

如何给出 A::f 中调用 func 的参数？

在 C++的类定义中引进了 this 指针变量，每个成员函数中都有一个隐含的形参 this，其类型为相应类的指针常量（对于上述的类 A，this 的类型为 A＊const）。当一个成员函数被调用时，编译系统自动把调用的对象地址传给 this。例如，对于 a.f()，编译器将把它编译成：

```
A::f(&a);
```

因此，在 A::f 中调用 func 时，可把 this 作为参数调用 func，即：func(this)，就能实现所要求的功能。

实际上，在成员函数中访问类数据成员时，应该写成：

this-><数据成员名>

一般情况下，this->可以省略，但要把对象作为整体来访问时必须显式给出 this。

3、与面向对象的 C++程序等价的面向过程的 C++程序

一般来说，一个面向对象的 C++程序可以转换成功能等价的面向过程程序来理解。例如，对于下面的面向对象的 C++程序：

```
class A
{   int x,y;
   public:
     void f();
     void q(int i) { x = i; f(); }
};
......
A a,b;
a.f();
a.q(1);
b.f();
b.q(2);
```

可转换成下面的功能等价的面向过程的 C++程序：

```
struct A
{ int x,y;
};
void A_f(A *const this);
void A_q(A *const this, int i)
{ this->x = i;
   A_f(this);
}
......
A a,b;
A_f(&a);
A_q(&a,1);
A_f(&b);
A_q(&b,2);
```

上述的程序虽然是用 C++中过程化的语言成分书写的，但所体现的却是面向对象的思想。因此，用过程式语言也能进行面向对象程序设计，不过，程序设计者要做很多额

外的工作。

6.2.4 成员的访问控制：信息隐藏

在C++的类定义中，可以用访问控制修饰符public,private和protected对类成员的访问进行限制。

1、public

对public成员的访问不受限制。

2、private

private成员只能在本类和友元中访问。

3、protected

protected成员只能在本类、派生类和友元中访问。

例如：

```
class A
{ public:
    int x;
    void f() { 允许访问：x,y,z,f,g,h }
  private:
    int y;
    void g() { 允许访问：x,y,z,f,g,h }
  protected:
    int z;
    void h() { 允许访问：x,y,z,f,g,h }
};
......
A a;
a.x = 1; //OK
a.f(); //OK
a.y = 1; //Error
a.g(); //Error
a.z = 1; //Error
a.h(); //Error
```

在类定义中，可以有多个public、private和protected访问控制声明，C++的默认访问控制是：private，如：

```
class A
{   int m,n; //m,n 的访问控制属性为 private
  public:
    int x;
    void f();
  private:
    int y;
    void g();
  protected:
    int z;
    void h();
  public:
    void f1();
};
```

一般情况下，类的数据成员应该指定为 private 或 protected，在类的内部使用的成员函数也应该指定为 private 或 protected，只有提供给外界使用的成员函数才指定为 public。属性为 public 的成员构成了类的接口（interface）。

例：用链表重新实现前面的类 Stack。

```
#include <iostream.h>
#include <stdio.h>
class Stack
{ public:
      Stack() { top = NULL; }
      bool push(int i);
      bool pop(int& i);
  private:
      struct Node
      { int content;
        Node *next;
      } *top;
};
bool Stack::push(int i)
{ Node *p=new Node;
  if (p == NULL)
  { cout << "Stack is overflow.\n";
    return false;
  }
  else
```

```
    { p->content = i;
      p->next = top;
      top = p;
      return true;
    }
}
bool Stack::pop(int& i)
{ if (top == NULL)
    { cout << "Stack is empty.\n";
      return false;
    }
    else
    { Node *p=top;
      top = top->next;
      i = p->content;
      delete p;
      return true;
    }
}
……
Stack st1,st2;
int x;
st1.push(12); st1.pop(x);
st2.push(20); st2.pop(x);
```

6.2.5 成员函数的重载

一个类的成员函数是可以重载的，它遵循一般函数的重载规则。

例：

```
class A
{ public:
      void f();
      int   f(int i);
      double f(double d);
};
......
void main()
{ A a;
  a.f();
  a.f(1);
```

```
    a.f(1.0);
}
```

6.3 构造函数和析构函数

6.3.1 构造函数

1、对象的初始化

对象的初始化是指对象数据成员的初始化。在创建对象后、使用对象前，往往要对对象的某些数据成员进行初始化。

要初始化对象的成员变量，一种方法是直接赋值，但是，由于成员变量一般为私有的(private)，直接赋值往往不行，如：

```
class A
{ int x;
  ......
};
......
A a;
a.x = 0; //Error, x 是私有成员
```

另一种初始化对象成员变量的方法是：在对象类中提供一个实现对象初始化的普通成员函数(如：init_obj)，创建对象后，通过调用该函数来初始化对象。例如：

```
class A
{   int x;
  public:
    void init_obj() { x = 0; }
  ......
};
......
A a;
a.init_obj();
```

该方法的缺点是：会带来使用上的不便(使用对象前必须显式调用该函数)和不安全(未调用初始化函数就使用对象)。

另外，上述方法无法解决对常量和引用数据成员的初始化问题。在面向对象程序设计中，用类的构造函数来解决对象初始化问题。

2、构造函数的定义

构造函数是指：在对象类中定义或声明的与类同名、无返回类型的函数。当创建对象时，构造函数将被自动调用。对构造函数的调用是对象创建过程的一部分，不能在其它地方调用构造函数。例如：

```
class A
{   int x;
  public:
    A() { x = 0; } //构造函数
   ......
};
......
A a; //创建对象 a，并调用 a 的构造函数 A()。
a.A(); //Error
```

构造函数可以重载，其中，不带参数的构造函数被称为默认构造函数。当对象类中未提供任何构造函数时，编译程序将为之提供一个默认构造函数。如果对象类中提供了构造函数，但没有提供默认构造函数，编译程序将不再为其提供默认构造函数。

类的构造函数一般是公开的(public)，但有时也声明为私有的，其作用是限制创建该类对象的范围，只能在本类和友元中创建该类对象。

3、构造函数的调用

在创建对象时，构造函数将自动被调用，所调用的构造函数在创建对象时指定。

例：

```
class A
{    …
  public:
     A();
     A(int i);
     A(char *p);
};
A a1; ⇔ A a1=A(); //调 A()，注意：不能写成：A a1();
A a2(1); ⇔ A a2=A(1); ⇔ A a2=1; //调 A(int i)
A a3(“abcd”); ⇔ A a3=A(“abcd”); ⇔ A a3=“abcd”; //调 A(char *)

A a[4]; //调用 a[0]、a[1]、a[2]、a[3]的 A()
A b[5]={A(),A(1),A("abcd"),2,"xyz"};   //调用 b[0]的 A()、b[1]的 A(int)、
                                  //b[2]的 A(char *)、b[3]的 A(int)和 b[4]的 A(char *)。
```

4、成员初始化表

在定义构造函数时，函数头和函数体之间可以加入一个对数据成员进行初始化的表，用于对数据成员进行初始化，特别是对常量和引用数据成员进行初始化。

例：

```
class A
{   int x;
    const int y;
    int& z;
  public:
    A(): y(1),z(x), x(0) { … }
};
```

注：对 x 的初始化可以放在成员初始化表中，也可以在构造函数体中进行赋值。

数据成员初始化的次序取决于它们在类定义中的声明次序，与它们在成员初始化表中的次序无关。

6.3.2 析构函数

析构函数是名为：~<类名>，无参数和无返回类型的成员函数。在对象消亡时，析构函数将会自动被调用。例如：

```
class A
{   int x;
  public:
    A();
    ~A(); //析构函数
      ......
};
void f()
{ A a; //调用 a 的构造函数 A()。
   ......
} //调用 a 的析构函数~A()。
```

析构函数的作用是归还对象申请的资源。

例：

```
#include <string.h>
class String
{     char *str;
```

```
public:
   String()
   { str = NULL;
   }
   String(char *p)
   { str = new char[strlen(p)+1];
      strcpy(str,p);
   }
   ~String()
   { if (str != NULL) delete []str;
      str = NULL;
   }
   int length()
   { return strlen(str);
   }
   char get_char(int i)
   { return str[i];
   }
   void set_char(int i, char value)
   { str[i] = value;
   }
   char &char_at(int i)
   { return str[i];
   }
   char *get_str()
   { return str;
   }
   char *strcpy(char *p)
   { delete []str;
      str = new char[strlen(p)+1];
      strcpy(str,p);
      return str;
   }
   String &strcpy(String &s)
   { delete []str;
      str = new char[strlen(s.str)+1];
      strcpy(str,s.str);
      return *this;
   }
   char *strcat(char *p);
```

```
        String &strcat(String &s);
};
void main()
{ String s1;
  String s2("abcdefg");
  s1.strcpy("xyz");
  s2.strcpy(s1);
  for (int i=0; i<s1.length(); i++)
  { if (s1.get_char(i) >= 'a' && s1.get_char(i) <= 'z')
        s1.set_char(i,s1.get_char(i)-'a'+'A');
  }
  for (i=0; i<s2.length(); i++)
  { if (s2.char_at(i) >= 'a' && s2.char_at(i) <= 'z')
        s2.char_at(i) = s2.char_at(i)-'a'+'A';
  }
  cout << s1.get_str() << endl << s2.get_str() << endl;
}
```

析构函数可以显式调用。

6.3.3 成员对象的初始化

类的数据成员可以是另一个类的对象。

1、初始化

成员对象的初始化在包含该成员对象的对象类中指定。默认情况下调用成员对象的默认构造函数，如需要调用成员对象的非默认构造函数，则需要在包含该成员对象的对象类构造函数的成员初始化表中指定。

例：

```
class A
{    int m;
  public:
     A() { m = 0; }
     A(int m1) { m = m1; }
};
class B
{    int x;
     A a;
  public:
     B() { x = 0; }
```

```
        B(int x1) { x = x1; }
        B(int x1, int m1): a(m1) { x = x1; }
};
void main()
{ B b1; //调用 B::B()和 A::A()
  B b2(1); //调用 B::B(int)和 A::A()
  B b3(1,2); //调用 B::B(int,int)和 A::A(int)
  ……
}
```

2、构造函数与析构函数的调用次序

当创建包含对象成员的对象时，先调用成员对象的构造函数，再调用本身对象的构造函数。当包含对象成员的对象消亡时，先调用本身对象的析构函数，再调用成员对象的析构函数。

6.3.4 拷贝构造函数

当创建一个对象时，如果用另一个同类的对象对其初始化，则会调用一个特殊的构造函数：拷贝构造函数。

```
class A
{    int x,y;
  public:
     A();
     A(const A& a); //拷贝构造函数，const 可有可无
};
```

有三种情况将调拷贝构造函数：

情形 1：

```
A a1; //调默认构造函数对 a1 进行初始化
A a2(a1);  或  A a2=a1;  或  A a2=A(a1); //调拷贝构造函数对 a2 进行初始化
```

情形 2：

```
f(A x);
......
A a;
f(a); //创建对象 x，调拷贝构造函数用对象 a 对其初始化。
```

情形 3：

```
A f()
{ A a;
   ……
   return a; //创建一个 A 类的临时对象，调拷贝构造函数用对象 a 对其初始化。
}
.....
f();
```

注意：上述情形 1 对拷贝构造函数的调用比较明显，而情形 2 和 3 对拷贝构造函数的调用就不是那么明显。

拷贝构造函数用于实现如何用已存在的一个对象对正在创建的对象进行初始化，如：

```
class A
{    int x,y;
   public:
      A() { x = 0; y = 0; }
      A(const A& a)
      { x = a.x;
         y = a.y;
      }
}
......
A a1;
A a2(a1);
```

对上述的对象 a2，将调用类 A 的拷贝构造函数来对其初始化，初始化的结果为：a2.x 和 a2.y 分别初始化为 a1.x 和 a1.y 的值。由于拷贝构造函数是自己定义的，所以任何的拷贝行为都可以实现，如：

```
class A
{    int x,y;
   public:
      A();
      A(const A& a)
      { x = a.x+1;
         y = a.y+1;
      }
}
```

```
.....
A a1;
A a2(a1);
```

上述的对象 a2 初始化为：a2.x 和 a2.y 的值分别为 a1.x 和 a1.y 的值加 1。

如果类定义中包含有对象成员，则对象成员的拷贝可以由成员对象类的拷贝构造函数来实现，这时，必须要在类定义中显式地指出。如：

```
class A
{    int x,y;
   public:
      A();
      A(const A& a)
      { x = a.x+1;
         y = a.y+1;
      }
};
class B
{   int z;
     A a;
   public:
     B();
     B(const B& b): a(b.a) //调 A 的拷贝构造函数用 b.a 对 a 进行初始化
      { z = b.z;
      }
}
.....
B b1;
B b2(b1);
```

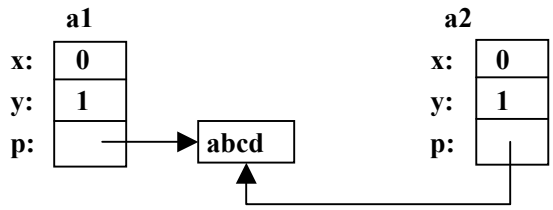
如果在类定义中没有给出拷贝构造函数，则编译系统提供一个默认拷贝构造函数，该拷贝构造函数的行为是：逐个成员拷贝(member-wise copy)。对于非对象成员，采用逐位拷贝（bit-wise copy)；对于对象成员，则调用对象成员类的拷贝构造函数来实现对象成员的拷贝。注意，则该定义是递归的。

一般情况下，默认拷贝构造函数的行为足以满足要求，为什么还要显式定义拷贝构造函数呢？

例子：

```
class A
```

```
{    int x,y;
     char *p;
  public:
     A(char *str)
     { x = 0;
       y = 1;
       p = new char[strlen(str)+1];
       strcpy(p,str);
     }
     ~A() { delete []p; }
}
.....
A a1("abcd");
A a2(a1);
```

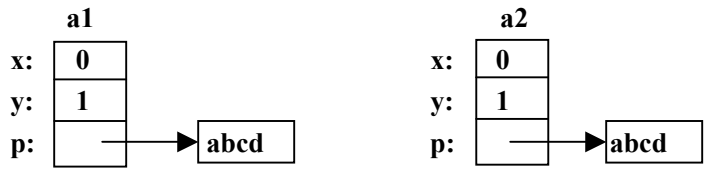


对象 a1 与 a2 的成员变量 p 指向同一个区域，修改一个对象将影响另一个。当对象 a1 和 a2 消亡时，同一块内存区域将被归还两次，导致程序运行异常。

解决办法：定义拷贝构造函数。

```
A::A(const A& a)
{ x = a.x;
  y = a.y;
  p = new char[strlen(a.p)+1];
  strcpy(p,a.p);
}
```

这时，a1.p 和 a2.p 将指向不同的区域：



6.4 友元

在一个类的外部不能访问该类的 private 成员，如需访问，则须通过该类的 public 成员来进行，这个限制有时会降低对 private 成员的访问效率，缺乏灵活性。

例：给出一个矩阵类(Matrix)、一个向量类(Vector)和一个全局函数(multiply)，全局函数实现矩阵和向量相乘。

```
#include <iostream.h>
#include <stdlib.h>
class Matrix
{   int *p_data;
    int lin,col;
  public:
    Matrix(int l, int c)
    { if (l <= 0 || c <= 0)
       { cerr << "Bad matrix size.\n";
          exit(1);
       }
       lin = 1;
       col = c;
       p_data = new int[lin*col];
    }
    ~Matrix()
    { delete []p_data;
    }
    int &element(int i, int j)
    { if (i < 0 || i >= lin || j < 0 || j >= col)
       { cerr << "Matrix index out of range\n";
          exit(1);
       }
        return *(p_data+i*col+j);
    }
    void dimension(int &l, int &c)
    { l = lin;
       c = col;
    }
    void display()
    { int *p=p_data;
```

```
        for (int i=0; i<lin; i++)
        { for (int j=0; j<col; j++)
            { cout << *p << ' ';
              p++;
            }
            cout << endl;
        }
      }
};
class Vector
{   int *p_data;
    int num;
  public:
    Vector(int n)
    { if (n <= 0)
      { cerr << "Bad vector size.\n";
        exit(1);
      }
      num = n;
      p_data = new int[num];
    }
    ~Vector()
    { delete []p_data;
    }
    int &element(int i)
    { if (i < 0 || i >= num)
      { cerr << "Vector index out of range.\n";
        exit(1);
      }
      return p_data[i];
    }
    void dimension(int &n)
    { n = num;
    }
    void display()
    { int *p=p_data;
      for (int i=0; i<num; i++,p++)
        cout << *p << ' ';
      cout << endl;
    }
```

```
};
void multiply(Matrix &m, Vector &v, Vector &r)
{ int lin,col;
   m.dimension(lin,col);
   for (int i=0; i<lin; i++)
   { r.element(i) = 0;
      for (int j=0; j<col; j++)
         r.element(i) += m.element(i,j)*v.element(j);
   }
}
void main()
{ Matrix m(10,5);
   Vector v(5);
   Vector r(10);
   ......
   multiply(m,v,r);
   m.display();
   v.display();
   r.display();
}
```

上述的函数 multiply 中多次通过调用成员函数 element 访问 m,v,r 的元素，每一次调用都要检查下标的合法性，因此效率不高。

在一个类定义中，可以指定某个全局函数、某个其它类或某个其它类的某个成员函数可以访问该类的私有和保护成员，即友元函数、友元类和友元类成员函数。

例：

```
......
void func() {…}
class A
{   …
   friend void func();   //友元函数
   friend class B;       //友元类
   friend void C::f(); //友元类成员函数，假定 void f()是类 C 的成员函数
};
```

友元的作用在于提高面向对象程序设计的灵活性，是数据保护和对数据的存取效率之间的一个折中方案。

注意：友元不具有传递性，即，假设 B 是 A 的友元、C 是 B 的友元，如果没有显式指

出 C 是 A 的友元，则 C 不是 A 的友元。

如果把函数 multiply 作为类 Matrix 和 Vector 的友元函数，在函数 multiply 中直接存取它们的私有成员，将会大大提高效率：

```
class Matrix
{ ......
   friend void multiply(Matrix &m, Vector &v, Vector &r);
};
class Vector
{ ......
   friend void multiply(Matrix &m, Vector &v, Vector &r);
};
void multiply(Matrix &m, Vector &v, Vector &r)
{ if (m.col != v.num || m.lin != r.num)
   { cerr << "Sizes of matrix and vector do not match.\n";
      exit(1);
   }
   int *p_m=m.p_data,*p_v,*p_r=r.p_data;
   for (int i=0; i<m.lin; i++)
   { *p_r = 0;
      p_v = v.p_data;
      for (int j=0; j<m.col; j++)
      { *p_r += (*p_m)*(*p_v);
         p_m++;
         p_v++;
      }
      p_r++;
   }
}
```

6.5 动态对象

在程序的堆(Heap)中创建的对象称为动态对象。动态对象用 new 操作符创建，用 delete 操作符撤消。

1、单个动态对象的创建与撤消

```
class A
{ …
  public:
```

```
    A();
    A(int);
};
…
A *p,*q;
p = new A; // 首先在程序的堆中申请一块大小为 sizeof(A)的
// 空间，然后调用 A 的默认构造函数对该空间上的对
// 象初始化，最后返回创建的对象的地址并赋值给 p。
q = new A(1); // 与上一条类似，不同之处在于：不是调用 A 的
// 默认构造函数，而是调用 A 的另一个构造函数：
// A::A(int)。
…
delete p; // 首先调用 p 所指向的对象的析构函数，然后释放对
// 象空间。
delete q; // 与上一条相同。
```

动态对象也可以采用 C++的库函数 malloc 来创建和 free 来撤消：

```
p = (A *)malloc(sizeof(A));
free(p);
```

但是，malloc 不调用 A 的构造函数，free 不调 A 类的析构函数，因此，对象的初始化和结束处理只能采用其它方式实现。

另外，new 操作符的优越性还在于：

(1) 自动分配足够的空间

(2) 自动返回指定类型的指针

(3) 可以重载

2、动态对象数组的创建与撤消

```
A *p;
p = new A[100];
delete []p;
```

new 的一般形式：

```
new <class_name>[size_1][size_2]…[size_n]
```

注意：

(1) 不能显式地初始化对象数组，相应的类必须有默认构造函数。

(2) delete 中的[]不能省。

6.6 const 成员

1、const 成员函数

问题：对下面的类定义 **A** 和创建的常量对象 **a:**

```
class A
{   int x,y;
 public:
    void f();
};
const A a;

a.f(); //是否合法？
```

因为 a 是常量，在程序中不应改变 a（改变 a 的成员变量的值）。直觉上，如果 A::f 中没有改变 a 的成员变量的值，则是合法的，否则是不合法的。

但是，编译程序往往无法知道函数 f 中是否改变了 a 的成员变量的值（为什么？），为了安全，编译程序认为 a.f();是不合法的。

因此，要使得 a.f();合法，不仅在 A::f 中不能改变 a 的成员变量的值，而且要在 f 的定义和声明处显式地指出之：

```
class A
{ …
  void f() const;
};
void A::f() const
{ …
}
```

const 成员函数不能修改成员变量的值，它往往用于只使用成员变量的值而不改变成员变量值的成员函数。

成员函数加上 const 修饰符的作用有两个：

(1) 对使用（调用）**const** 成员函数的地方，告诉编译器该成员函数不会改变对象成员变量的值。

(2) 对 **const** 成员函数定义的地方，告诉编译器该成员函数不应该改变对象成员变量的

值。

2、const 成员变量

在数据成员的声明中加上 const，则表示是常量：

```
class A
{ const int x;
}
```

常量成员的初始化放在构造函数的成员初始化表中进行。

6.7 静态成员

对象是类的实例，类刻划了一组具有相同属性的对象。

类中声明的成员变量属于实例化后的对象，有多个拷贝（拷贝个数由创建的对象个数决定）。

问题：同一个类的不同对象如何共享变量？如果把这些共享变量定义为全局变量，则缺乏数据保护，因为其它类的对象往往也能存取这些全局变量。

1、静态成员变量

```
class A
{ int x,y;
   static int shared;
   .....
};
int A::shared=0;

A a,b;
```

类定义中声明的静态变量被该类的对象所共享，即，对该类的所有对象，类的静态成员变量只有一个拷贝。静态变量也遵循类的访问控制。

2、静态成员函数

```
class A
{ static int shared;
   int x;
  public:
   static void f() { …shared…}
   void q() { …x…shared…}
};
```

静态成员函数只能存取静态成员变量和调用静态成员函数。静态成员函数也遵循类的访问控制。

3、静态成员的使用

(1) 通过对象使用

```
A a;
a.f();
```

(2) 通过类使用

```
A::f();
```

4、C++对类也是对象的观点的支持

在一些面向对象程序设计语言（如：Smalltalk）中，把类也看成是对象，其属性由元类（meta class）来描述。C++也支持这个观点，类定义中的静态成员就是属于类这个对象，对该对象的成员的访问可以通过类名加上::来进行。

例：

```
class A
{   static int obj_count;
    …
  public:
    A() { obj_count++; }
    ~A() { obj_count--; }
    static int get_num_of_obj() { return obj_count; }
    …
};
int A::obj_count=0;
……
```

对上面定义的类 A，如果把它看成对象的话，它有两个成员：obj_count 和 get_num_of_obj，分别用于表示程序运行的某一时刻存在多少个 A 类的对象和获取 A 类对象的数目。

第7章 运算符重载

7.1 需要性

运算符重载能够提高语言的灵活性和可扩充性，实现多态。C++语言提供的操作符只定义了对基本数据类型的运算，操作符重载机制提供用已有的操作符来对自己定义的数据类型进行操作的手段，从而使得程序更容易理解。

例：

```
class Complex
{   double real, imag;
  public:
    Complex() { real = 0; imag = 0; }
    Complex(double r, double i) { real = r; imag = i; }
    Complex add(Complex& x)
    { Complex temp;
      temp.real = real+x.real;
      temp.imag = imag+x.imag;
      return temp;
    }
};
……
Complex a(1,2),b(3,4),c;
c = a.add(b);
```

上述程序段实现了两个复数的相加操作，如果能表示成：a = b + c，将会更加容易理解。

利用操作符重载机制可以把上述的 Complex 类重新定义成：

```
class Complex
{   double real, imag;
  public:
    Complex() { real = 0; imag = 0; }
    Complex(double r, double i) { real = r; imag = i; }
    Complex operator + (Complex& x)
    { Complex temp;
      temp.real = real+x.real;
      temp.imag = imag+x.imag;
      return temp;
    }
};
```

或

```
class Complex
{   double real, imag;
  public:
    Complex() { real = 0; imag = 0; }
    Complex(double r, double i) { real = r; imag = i; }
    friend Complex operator + (Complex& c1, Complex& c2);
};
Complex operator + (Complex& c1, Complex& c2)
{ Complex temp;
  temp.real = c1.real + c2.real;
  temp.imag = c1.imag + c2.imag;
  return temp;
}
```

对 Complex 类重新定义后，Complex 对象就能用操作符+进行运算：

```
Complex a(1,2),b(3,4),c;
c = a + b;
```

对 C++操作符进行重载，可以实现按照 C++表达式的语法来描述对象的一些操作，如：用操作符==、!=、<、<=、>、>=等来对两个对象进行比较等。对用户自定义的对象，C++只预定义三个操作符：=（赋值）、.（取成员）和&（取地址）。

1、操作符重载的基本原则

(1) 操作符的重载或者作为类的成员函数或者是带有类参数的全局函数。

(2) 遵循已有操作符的语法：单目/双目，优先级，结合性。

(3) 遵循已有操作符的语义（不是必需的）。如果某个已有操作符的语义对某类对象不是很明显，则不要重载它，用普通的成员函数来实现更容易理解，如：对字符串类重载++和--以实现字符串的大小写转换会给理解带来麻烦，用成员函数 **upper** 和 **lower** 更有利于理解。

2、可重载的操作符

除：.，.*，::，?:四个外，其它操作符都可以重载。

关于操作符：.*和->*

(1) a.*b：a 是一个对象，b 是一个类成员指针。含义是：访问对象 a 的由 b 指向的成员。

(2) a->*b：a 是一个对象指针，b 是一个类成员指针。含义是：访问 a 所指向对象的由

b 指向的成员。

例：

```
class A
{ public:
    int x;
    void f();
};
......
int A::*pm_x = &(A::x);
void (A::*qm_f)() = &(A::f);

A a;
A *p=new A;

a.x = 0;
a.*pm_x = 0;

a.f();
(a.*qm_f)();

p->x = 0;
p->*pm_x = 0;

p->f();
(p->*pm_f)();
```

7.2 双目操作符重载

需要两个参数，可以作为类成员函数或全局（友元）函数。

7.2.1 作为类成员函数

第一个参数是 this，它是隐含的，不需要给出，只需要给出第二个参数。

格式：

(1) 声明

```
class <class name>
{   ……
```

```
    <ret type> operator # (arg);
};
```

(2) 定义

```
<ret type> <class name>::operator #(arg) { … }
```

(3) 使用

```
<class name> a,b;
a # b, 或，a.operator#(b)
```

例如：

```
class Complex
{   ......
   public:
      bool operator ==(const Complex& x)
      { return (real == x.real) && (imag == x.imag);
      }
};
......
Complex a(1,2),b(3,4);
......
if (a == b) //或者  if (a.operator ==(b))
......
```

7.2.2 作为全局（友元）函数

需要给出两个参数，其中至少有一个类型为类。

格式：

(1) 声明

```
class <class name>
{ ..…
  friend <ret type> operator # (arg1,arg2);
};
```

(2) 定义

```
<ret type> operator #(arg1,arg2) { … }
```

限制：=，()，[]和→不能作为全局（友元）函数重载。

(3) 使用

```
<class name> a,b;
a # b
```

例如：

```
class Complex
{   ......
   public:
      firend bool operator < (const Complex& x, const Complex& y);
};
bool operator < (const Complex& x, const Complex& y)
{ return x.real*x.real + x.imag*x.imag < y.real*y.real + y.imag*y.imag;
}
......
Complex a(1,2),b(3,4);
......
if (a < b)
......
```

7.2.3 作为类成员函数和作为全局(友元)函数的区别

(1) 成员函数只需给一个参数，而全局（友元）函数必须给两个参数

(2) 有时必须用全局（友元）函数重载操作符。

例如：重载操作符+，实现实数与复数的混合运算。

```
class Complex
{   ......
   public:
      firend Complex operator + (const Complex& c1, const Complex& c2);
      firend Complex operator + (double d, const Complex& c);
      firend Complex operator + (const Complex& c, double d);
};
Complex operator + (const Complex& c1, const Complex& c2)
{ Complex temp;
```

```
    temp.real = c1.real + c2.real;
    temp.imag = c1.imag + c2.imag;
    return temp;
}
Complex operator + (double d, const Complex& c)
{ Complex temp;
    temp.real = d + c.real;
    temp.imag = c.imag;
    return temp;
}
Complex operator + (const Complex& c, double d)
{ Complex temp;
    temp.real = c.real + d;
    temp.imag = c.imag;
    return temp;
}
......
Complex a(1,2),b(3,4),c1,c2,c3;
c1 = a + b;
c2 = 10.2 + a;
c3 = b + 21.5;
```

7.3 单目操作符重载

重载单目操作符时需要给出一个参数。

7.3.1 作为类成员函数

参数是 this，它是隐含的，不需要给出。

格式：

(1) 声明

```
class <class name>
{ …
      <ret type> operator # ();
};
```

(2) 定义

```
<ret type> <class name>::operator #() { … }
```

(3) 使用

```
<class name> a;
#a, 或，a.operator#()
```

例如：

```
class Complex
{   ......
   public:
      Complex operator -()
      { Complex temp;
         temp .real = -real;
         temp.imag = -imag;
         return temp;
      }
};
......
Complex a(1,2),b;
b = -a;
```

问题：怎么处理后置的操作符++和--？

对于单目操作符，重载的是其前置用法。操作符++和--有前置和后置两种用法，如果没有特别说明，前置和后置均采用同一个重载函数来解释。为了能够区分前置及后置的++和--的重载，C++中采用为后置的重载函数增加一个参数来区分它们。

例如：

```
class Counter
{   int value;
   public:
      Counter() { value = 0; }
      Counter& operator ++() //  前置的++
      { value++;
         return *this;
      }
      Counter operator ++(int x) //后置的++
      { Counter temp=*this;
         value += x;
         return temp;
      }
```

```
};
.....
Counter a,b,c;
b = ++a; //使用的是上述类定义中第一个重载的++
c = a++; //使用的是上述类定义中第二个重载的++，把 1 传给形参 x
```

如果未定义第二个重载，则 a++用第一个定义；如果定义了第二个重载，则 a++用第二个定义。对于第二个重载，还可以采用形式：a.operator++(i)来使用之，这时，形参 x 将得到 i。

注意：上述的第一个重载返回的是引用（为什么？）。

7.3.2 作为全局（友元）函数

需要给出一个参数。

格式：

(1) 声明

```
class <class name>
{ …
  friend <ret type> operator # (arg);
};
```

(2) 定义

```
<ret type> <class name>::operator #(arg) { … }
```

(3) 使用

```
<class name> a;
#a
```

例如：

7.4 几个特殊操作符的重载

7.4.1 赋值操作符=

只能作为非静态成员函数重载，不能继承。

对每个类，C++都定义了一个默认赋值操作符=重载函数，其行为是：逐个成员赋值

(member-wise assignment)，对包含有对象成员的类，该定义是递归的。

有时，默认赋值操作符=函数不能满足需要（特别是类包含指针成员时），这时需要自己重载=。

例：

```
class A
{    int x,y;
     char *p;
  public:
     A() { p = NULL; }
     A(char *str)
     { p = new char[strlen(str)+1];
       strcpy(p,str);
     }
     ~A()
     { delete []p;
       p = NULL;
     }
     A& operator = (const A& a)
     {   if (this == &a) return *this;
          x = a.x; y = a.y;
          delete []p;
          p = new char[strlen(a.p)+1];
         strcpy(p,a.p);
         return *this;
     }
};
```

如果上述类 A 中没有给出重载的=，则对于下面的语句，将会使得 a.p 和 b.p 指向同一块区域。

```
A a,b("abcdefg");
.......
a = b;
```

注意：拷贝构造函数和赋值操作符=重载的区别：

(1) 在创建一个对象时用另一个已存在的同类对象对其初始化，调用拷贝构造函数；

(2) 对两个已存在的对象，用其中一个对象来改变另一个对象的状态，调用赋值操作符=重载。

例如：

```
A a;
A b=a; //调用拷贝构造函数用 a 对 b 进行初始化
......
b = a; //调用赋值操作符=重载用 a 对 b 进行赋值
```

实际上，对于赋值操作符重载函数来讲，它隐含着一个析构过程和一个拷贝构造过程。

7.4.2 数组元素访问运算符[]

只能作为非静态成员函数重载。

例：

```
class string
{   char *p;
  public:
    string(char *p1)
    { p = new char [strlen(p1)+1];
      strcpy(p,p1);
    }
    char& operator [](int i)
    { return p[i];
    }
}
…
string s(“abcd”);
cout << s[2];
s[2] = 'z';
```

7.4.3 成员访问运算符->

只能作为非静态成员函数重载，用于实现 smart pointers。

->为二元运算符，如果重载则当作一元操作符来用，仔细体会下面的例子。

例：在程序执行中的任意时刻，获得对一个对象的访问次数。

```
class A
{   int x,y;
  public:
    void f();
    void g();
};
```

```
class B
{    A *p_a;
     int count;
  public:
     B(A *p)
     { p_a = p;
        count = 0;
     }
     A *operator ->()
     { count++;
        return p_a;
     }
     int num_of_a_access()
     { return count;
     }
};
…
A a;
B b(&a);
b->f(); //等价于：b.operator->()->f(); 即访问的是 a.f()
b->g(); //等价于：b.operator->()->g(); 即访问的是 a.g()
//上述两个->是重载的->。
cout << b.num_of_a_access(); //  显示对对象 a 的访问次数

B *p=&b;
p->num_of_a_access(); //  显示对对象 a 的访问次数
p->f(); //Error, b 没有 f();
p->g(); //Error, b 没有 g()。
//上述三个->是未重载的->。
```

7.4.4 动态存贮分配与去配运算符 new 与 delete

只能作为静态成员函数重载。

1、为什么要重载 new 与 delete

对于频繁地在堆空间中创建和撤消对象的程序来讲，由于 new 与 delete 将调用系统的通用堆存储管理来进行动态内存的分配与去配，而系统的堆存储管理要考虑各种大小的堆内存分配，对某个特定大小的堆内存分配的效率往往是不高的。程序自己来管理

堆内存可以提高程序的效率。基本做法是：程序首先调用系统堆存储分配申请一块较大的内存空间，当程序需要创建和撤消动态对象时，可以在该内存中自己进行对象空间的存储分配与去配管理，这样，可以大大提高程序的效率。可以通过重载 new 与 delete 来自己实现堆内存的管理。

2、重载 new

在定义一个类 A 时，提供一个具有下述原型的成员函数：

void *operator new(size_t size,…);

该函数的名为 operator new，返回类型必须为 void *，第一个参数类型为 size_t（unsigned int），其它参数可有可无。

当用操作符 new 动态创建 A 类对象时，系统自动计算对象的大小并把它传给重载的 new 的第一个参数 size。如果重载的 new 有其它参数，则动态对象的创建采用以下形式：

A *p=new (…) A;

这里，…表示传给 new 的其它实参。

new 的重载可以有多个，在某类中重载了 new 后，通过 new 动态创建该类的对象时将不再调用内置的（预定义的）new 操作来分配内存空间，而是调用自己重载的 new 操作。

3、重载 delete

在定义一个类 A 时，提供一个具有下述原型的成员函数：

void operator delete(void *p, size_t size);

该函数的名为 operator delete，返回类型必须为 void，第一个参数类型为 void *，第二个参数可有可无，如果有，则必须是 size_t 类型。当用 delete 撤消一个对象时，系统把该对象的地址传给重载的 delete 的第一个形参 p；如果重载的 delete 有第二个参数，则系统会把欲撤消的对象的大小传给之。

delete 的重载只能有一个，在某类中重载了 delete 后，通过 delete 撤消对象时将不再调用内置的（预定义的）delete 操作，而是调用自己重载的 delete 操作。

重载的 new 和 delete 是静态成员，也遵循类的访问控制，并且可以继承。

例：

const int NUM=32;

```
class A
{    ......
   public:
      void *operator new(size_t size);
      void operator delete(void *p);
   private:
      static A *p_free;
      A *next
};
A *A::p_free=NULL;
void *A::operator new(size_t size)
{ A *p;
   if (p_free == NULL)
   { p_free = (A *)new char[size*NUM];
      for (p=p_free; p!=p_free+NUM-1; p++)
         p->next = p+1;
      p->next = NULL;
   }
   p = p_free;
   p_free = p_free->next;
   return p;
}
void A::operator delete(void *p, size_t size)
{ ((A *)p)->next = p_free;
   p_free = (A *)p;
}
```

7.4.5 类型转换运算符

1、带一个参数的构造函数用作类型转换

带一个参数的构造函数可以用作从一个基本数据类型或其它类到一个类的隐式转换。

例：

```
class Complex
{   double real, imag;
   public:
      Complex() { real = 0; imag = 0; }
      Complex(double r) { real = r; imag = 0; }
      Complex(double r, double i) { real = r; imag = i; }
```

```
        Complex operator + (Complex& x)
        { Complex temp;
          temp.real = real+x.real;
          temp.imag = imag+x.imag;
          return temp;
        }
};
......
Complex c1(1,2),c2,c3;
c2 = c1 + 1.7; //1.7 隐式转换成一个复数对象 Complex(1.7)
c3 = 2.5 + c2; //2.5 隐式转换成一个复数对象 Complex(2.5)
```

在上述的例子中，由于可以从一个 double 类型的数据隐式转换成一个 Complex 类的对象，因此，不必额外定义操作符+的两个参数类型分别为：（double, const Complex&）和(const Complex&, double)的全局重载函数就能实现 Complex 和 double 类型数据之间的混合+操作。

2、自定义类型转换

自定义的类型转换操作符可以实现从一个类到一个基本数据类型或一个其它类的转换。

例：

```
class A
{   int x,y;
  public:
    ......
    operator int() { return x+y; }
};
……
A a;
int i=1,z;
z = i+a;
```

当在一个类中同时定义具有一个参数（t 类型）的构造函数和 t 类型转换操作符时，会产生歧义问题，如：

```
class A
{   int x,y;
  public:
    A(int i) { x = i; y = 0; }
```

```
    operator int() { return x+y; }
    A operator +(const A& a)
     { A temp;
        temp.x = x + a.x;
        temp.y = y + a.y;
        return temp;
     }
};
……
A a;
int i=1,z;
z = a+i; //是 a 转换成 int，还是 i 转换成 A？
```

第8章 继承－－派生类

8.1 继承的概念

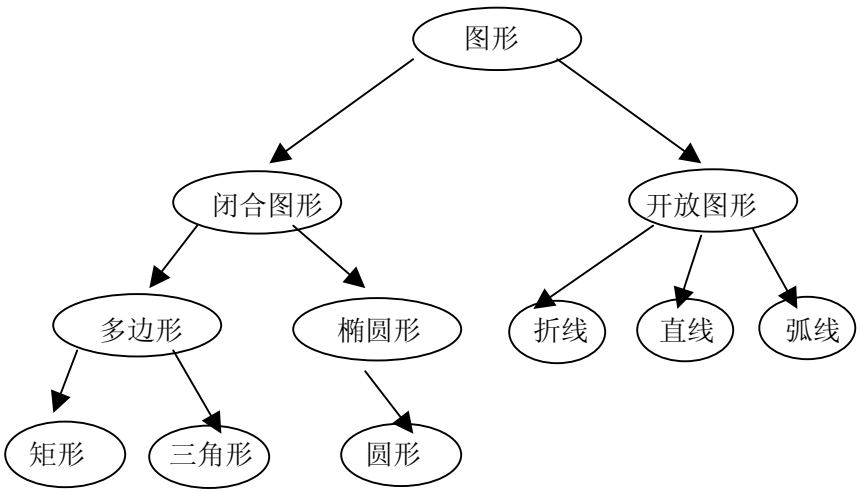
随着软件越来越复杂、软件开发越来越困难，软件复用越来越受到人们的重视。理想的软件复用是采用硬件的复用方式，即，在开发新软件时能够把现有软件或软件的一部分原封不动地拿过来用。

目前，由于种种原因（技术和资源等限制），对已有软件不加修改地直接使用往往是不可能的。已有软件的功能与新软件所需要的功能是有差别的，如果要复用，就必须解决这个差别。如何处理这个差别？一种途径是修改已有软件的源代码，这不仅需要要读懂源代码，而且也不可靠，另外，该途径是基于已有软件有源代码的情况。

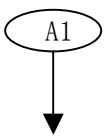
继承机制提供了另一种复用途径，它不需要修改已有软件，而是提供新功能的代码或对已有一些功能的重新实现，因此，继承是一种基于目标代码的代码复用机制。

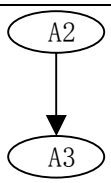
当然，继承机制除了支持软件复用外，它还具有：

- 对事物进行分类(派生类是基类的具体化)。
  把事物（概念）以层次结构表示出来，有利于描述和解决问题。

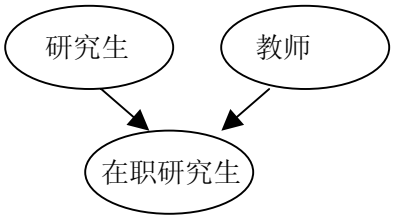


- 支持软件的增量开发(派生类是基类的延续)。
  软件的开发往往不是一次完成，它是随着对软件功能的逐步理解和改进不断完善。





- 对概念进行组合（多继承）。



在 C++中，继承是通过派生类来实现的。

8.2 基类与派生类（父类与子类）

对于下面的两个类 A 和 B，A 是基类，B 是派生类。

```
class A //基类
{  int x,y;
  public:
   f();
   g();
};
class B: public A //派生类
{  int z;
  public:
   h();
};
…
B b;
b.f(); //A 中的 f
b.h(); //B 中的 h
```

派生类对象除了拥有基类对象的所有成员外，还可以具有新的成员。例如：B 类的对象 b 具有成员变量 x、y 和 z 以及成员函数 f、g 和 h。

派生类的定义用于描述派生类与基类的差别，在 C++中，派生类中可以给出新的成员，也可以对基类的成员进行重定义：

```
class B: public A
{  int z;
  public:
   h();
   f();
};
…
b.f(); //B 中的 f。
```

对于基类的一个成员函数，如果派生类中没有定义与其同名的成员函数，则该成员函数在派生类的作用域内可见，否则，该成员函数在派生类的作用域内不可见，如果要使用之，必须用基类名受限：

```
b.A::f(); //A 中的 f
```

即使派生类中定义了与基类同名但参数不同的成员函数，基类的同名函数在派生类的作用域中也是不可见的：

```
class B: public A
{ …
  public:
    h() { f(1); A::f(); }
    f(int);
};
…
B b;
b.f(1); //Ok
b.f();  //Error
b.A::f(); //Ok
```

一个类可以用来定义（创建）对象，也可以用来定义派生类，即：一个类有两种用户：实例用户和派生类用户，在 C++类定义中为这两种用户分别提供了接口。

从分类的角度来讲，继承体现了类之间的一种一般与特殊的关系（is a kind of），基类往往代表一个一般概念，派生类代表一个具体概念，它是基类的一个特化。

继承不是代码复用的唯一方式，有些代码复用不宜用继承来实现，如飞机复用发动机的情形。类之间除了继承关系外，还存在一种整体与部分的关系（is a part of），这种关系称为聚集，即把一个类作为另一个类的成员对象类。

有人认为，从纯代码复用的角度来讲，聚集比继承更好，至少它可避免继承与封装的矛盾（在聚集关系中，一个类只有一种用户：实例用户），另外，继承的功能可以用聚集来实现。不过，继承更容易实现类型与子类型关系（参见 7.6 节）。

C++支持单继承（一个类只能有一个直接基类）和多继承（一个类可以有多个直接基类）。

8.3 单继承

8.3.1 定义

```
class 〈派生类名〉:[〈继承方式〉]〈基类名〉
{ 〈成员表〉
}
```

其中，继承方式可以是：public、private 和 protected。继承方式可以省略，默认为 private。

8.3.2 在派生类中对基类成员的访问

在派生类中可以使用基类的 public 成员，但不能使用基类的 private 成员。

在派生类中定义新的成员或对基类的成员重定义时往往需要用到基类的一些 private 成员，解决这个问题的一种办法是在基类中开放这些成员（声明为 public），但这样就带来一个问题：数据保护的破坏。当基类的内部实现发生变化时将会影响子类用户和基类的的实例用户。

为了缓解上述问题的严重性，在 C++中引进了 protected 成员保护控制，在基类中声明为 protected 的成员可以被派生类使用，但不能被基类的实例用户使用，这样缩小了修改基类的内部实现所造成的影响范围（只影响子类）。

另外，引进 protected 后，基类的设计者也会慎重地考虑应该把那些成员声明为 protected，至少应该把今后不太可能发生变动的、有可能被子类使用的、不宜对实例用户公开的成员声明为 protected。

8.3.3 继承方式

在 C++中，子类拥有基类的所有成员，问题是：基类的成员变成子类的什么成员：public、private 或 protected？即，基类的成员在派生类中具有何种访问控制？这是由继承方式来决定的。

1）public 继承

基类的 public 成员，在派生类成为 public 成员。

基类的 protected 成员，在派生类成为 protected 成员。

基类的 private 成员，在派生类成中不可直接用。

2）private 继承

基类的 public 成员，在派生类成为 private 成员。

基类的 protected 成员，在派生类成为 private 成员。

基类的 private 成员，在派生类成中不可直接用。

3）protected 继承

基类的 public 成员，在派生类成为 protected 成员。

基类的 protected 成员，在派生类成为 protected 成员。

基类的 private 成员，在派生类成中不可直接用。

继承方式决定了派生类的对象和派生类的派生类对基类成员的访问限制。

4）继承方式的调整

在任何继承方式中，除了基类的 private 成员，都可以在派生类中分别调整其访问控制，调整时采用：

〈基类名〉::〈基类成员名〉

具体规定视 C++的不同实现会有所不同，下面就 Visual C++作简单介绍。

例：

```
class A
{ public:
   void f1();
   void f2();
   void f3();
  protected:
   void g1();
   void g2();
   void g3();
};

class B:  A
{ public:
    A::f1;
    A::g1;
  protected:
```

```
        A::f2;
        A::g2;
      private:
        A::f3;
        A::g3;
    };
```

对基类一个重载成员函数名的访问控制的调整，将调整基类所有具有该名的重载函数。

```
class A
{ public:
    f();
    f(int);
  protected:
    f(float);
};
class B: public A
{ private:
    A::f; //A 类中的三个 f 均成为 private
};
```

派生类中如果定义了与基类同名的成员函数，则在派生类中不能再对基类中的同名函数进行访问控制调整，因为此时基类的同名函数在派生类的作用域中不可见。

```
class A
{ public:
    void f();
};
class B: A
{ public:
    void f(int);
    A::f; //Error
};

…
B b;
b.f(); /Error, no such member
```

8.3.4 派生类对象的初始化

派生类对象的初始化由基类和派生类共同完成：基类的数据成员由基类的构造函数初始化，派生类的数据成员由派生类的构造函数初始化。

当创建派生类的对象时，系统将会调用基类的构造函数和派生类的构造函数，构造函数的执行次序是：先执行基类的构造函数，再执行派生类的构造函数。如果派生类又有对象成员，则，先执行基类的构造函数，再执行成员对象类的构造函数，最后执行派生类的构造函数。析构函数的执行正好相反。

至于执行基类的什么构造函数，缺省情况下是执行基类的默认构造函数，如果要执行基类的非默认构造函数，则必须在派生类构造函数的成员初始化表中指出：

例：

```
class A
{  int x;
  public:
    A() { x = 0; }
    A(int i) { x = i; }
};
class B: public A
{   int y;
  public:
    B() { y = 0; }
    B(int i) { y = i; }
    B(int i, int j):A(i) { y = j; }
};

B b1; //执行A::A()和B::B()
B b2(1); //执行A::A()和B::B(int)
B b3(0,1); //执行A::A(int)和B::B(int,int)
```

8.3.5 单继承的例子

例1、一个公司中的职员和部门经理。

```
class Employee //职员类
{  int salary;
   char name[20];
  public:
   Employee(char *s, int n=0)
  { strcpy(name,s);
    salary = n;
```

```
  };
   void set_salary(int n) { salary = n; }
   int get_salary() { return salary; }
   char *get_name() { return name; }
};
define MAX_EMPLOYEE 20
class Manager: public Employee //部门经理
{  Employee *group[MAX_EMPLOYEE];
   int num_of_emp;
  public:
   Manager(char *s, int n=0): Employee(s,n)
    { num_of_emp = 0; }
   Employee *add_employee(Employee *e)
    { if (num_of_emp >= MAX_EMPLOYEE) return NULL;
      num_of_emp++;
      group[num_of_emp-1] = e;
      return e;
    };
   Employee *remove_employee(Employee *e)
    { for (int i=0; i<num_of_emp; i++)
        if (strcmp(e->get_name(),
                   group[i]->get_name()) == 0)
          break;
      if (i < num_of_emp)
      { for (int j=i+1; j<num_of_emp; j++)
          group[j-1] = group[j];
        num_of_emp--;
        return e;
      }
      else
        return NULL;
    }
};
......
Employee e1("Jack",1000),e2("Jane",2000);
Manager m("Mark",4000);
m.add_employee(&e1);
m.add_employee(&e2);
......
```

例 2、利用一个链表类实现一个队列类。队列是由若干个元素所组成的线性结构，具有两个操作：增加元素和删除元素，增加/删除的元素只能在线性结构的头或尾进行，并且一个在头，另一个在尾。

```
class NODE
{   NODE *next;
    int content;
  public:
    NODE(int i) { content = i; next = NULL; }
    int get_content() { return content; }
  friend class LINKED_LIST;
};
class LINKED_LIST
{   NODE *p_head; //表头指针
  public:
    LINKED_LIST() { p_head = NULL; }

    void insert_before(NODE *p, NODE *q); //将 p 所指向的结点
            //插入到表中 q 所指向的结点之前，如果 q 为 NULL，则
            //新结点插入到表头。
    void insert_after(NODE *p, NODE *q); //将 p 所指向的结点
            //插入到表中 q 所指向的结点之后，如果 q 为 NULL，则
            //新结点插入到表尾。
    void sort_insert(NODE *p); //将 p 所指向的结点插入到表中第
            //一个内容大于它的结点之前，如果没有内容大于它的
            //结点，则把它插在表尾
            。

    NODE *remove(NODE *p); //删除表中 p 所指向的结点
    NODE *remove_first();  //删除表中第一个结点
    NODE *remove_last();   //删除表中最后一个结点
    void remove_all();     //删除表中所有结点

    NODE *find_first(); //查找表中第一个结点
    NODE *find_last();  //查找表中最后一个结点
    NODE *find_next(NODE *p);//查找表中 p 所指结点的下一个结点
    NODE *find_prev(NODE *p);//查找表中 p 所指结点的上一个结点
    NODE *find(int i);  //查找表中第一个内容为 i 的结点
    NODE *find_next(NODE *p,int i);//查找表中 p 所指结点的下一
                                   //个内容为 i 的结点
    NODE *find_prev(NODE *p,int i);//查找表中 p 所指结点的上一
```

```
                                        //个内容为 i 的结点

    bool is_empty(); //判断表是否为空
    void sort(); //根据结点的内容（小->大）对结点进行排序
    void display(); //显示表中所有结点的内容
};
class QUEUE: LINKED_LIST
{ public:
    bool en_queue(int i)
    { NODE *p=new NODE(i);
      insert_after(p,NULL);
      return true;
    }
    bool de_queue(int &i)
    { NODE *p=remove_first();
      if (p == NULL) return false;
      i = p->get_content();
      delete p;
      return true;
    }
};
```

8.4 多继承

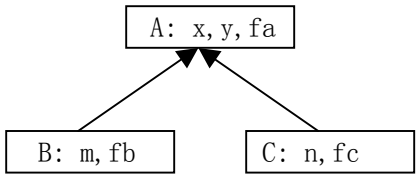
8.4.1 需要性

现有三个类：A、B 和 C，定义如下：

```
class A
{  int x,y;
  public:
   fa();
};
class B: public A
{  int m;
  public:
   fb();
};
class C: public A
{  int n;
  public:
```

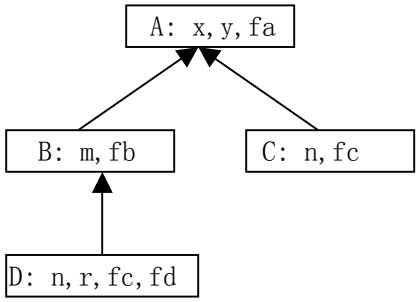
```
    fc();
};
```

用图可表示为：



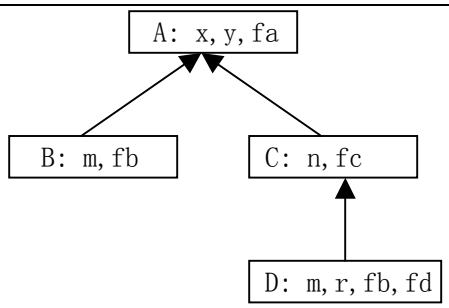
定义一个类 D，它具有类 A、B 和 C 的所有成员和新成员 r 和 fd，若采用单继承，则有两种做法：

1）
```
class D: public B
{  int n;
   int r;
  public:
   fc();
   fd();
};
```



或

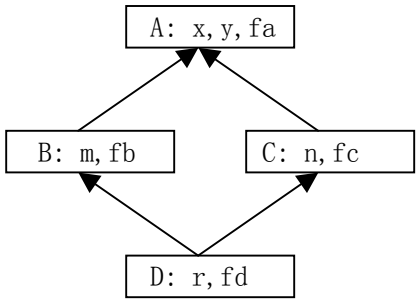
2）
```
class D: public C
{  int m;
   int r;
  public:
   fb();
   fd()
};
```



问题：1）概念不清，m 和 n 本来是独立的性质，现在具有层次关系了。
2）易造成不一致。

这时采用多继承比较合适：

```
class D: public B, public C
{  int r;
  public:
   fd();
};
```



8.4.2 定义

```
class 〈派生类名〉:[〈继承方式〉]〈基类名 1〉,[〈继承方式〉]〈基类名 2〉,…
{ 〈成员表〉
}
```

其中，继承方式可以是：public、private 和 protected。

1）继承方式及访问控制的规定同单继承。
2）派生类拥有所有基类的所有成员。
3）基类的声明次序决定：
a） 对基类构造函数/析构函数的调用次序

b) 对基类数据成员的存储安排

8.4.3 名冲突

用<基类名>::<基类成员名>解决。如：

```
class A
{ ......
  public:
   f();
   g();
};
class B
{ ......
  public:
   f();
   h();
};
class C: public A, public B
{ ......
  public:
   func()
   { A::f();
     g();
     B::f();
     h();
   }
};
......
C c;
c.func();
c.A::f();
c.g();
c.B::f();
c.h();
```

8.4.4 虚基类

在多继承中，如果直接基类有公共的基类，则，该公共基类中的成员变量在多继承的派生类中有多个副本。

例如：对下面的类 D：

```
class A
{  int x;
   …
};
class B: A;
class C: A;
class D: B, C;
```

则类 D 拥有两个 x 成员：B::x 和 C::x。如果要把这两个 x 合并为一个，则应把类 A 定义为虚基类：

```
class A;
class B: virtual A;
class C: virtual A;
class D: B, C;
```

注意：

1） 虚基类的构造函数由最新派生出的类的构造函数调用。

2） 虚基类的构造函数优先非虚基类的构造函数执行。

8.5 虚函数

8.5.1 类型相容

C++把类看作类型，有了类型就要考虑类型相容问题，其中包括赋值相容问题，即 a 和 b 是什么类型时，下面的赋值操作是合法的：

```
a = b;
```

1、在 C++中允许把派生类对象赋值给基类对象，这时赋值所产生的对象的身份已发生变化，对象已从派生类变成了基类，原属于派生类的属性已不存在。如：

```
class A
{ int x,y;
 public:
  f();
};
class B: public A
{  int z;
  public:
   f();
   g();
```

```
};
......
A a;
B b;
a = b; //OK,
a.f(); //A::f()
b = a; //Error
```

2、在C++中还允许基类的引用或指针可以引用或指向派生类对象，这时，引用或指向的对象身份没有发生变化，如：

```
A &r_a=b;  //OK
A *p_a=&b; //OK
B &r_b=a;  //Error
B *p_b=&a; //Error
```

对于2将会产生一种面向对象程序设计特有的多态，例如：

```
func1(A& a)
{ …
  a.f();
  …
}
func2(A *pa)
{ …
  pa->f();
  …
}
…
A a;
func1(a);
func2(&a);
B b;
func1(b);
func2(&b);
```

问题：对于func1(b)和func2(&b)，在func1和func2中的a.f()和pa->f()调用的是A类中的f还是B类中的f ？

8.5.2 前期绑定和后期绑定

对上述问题存在两种解决方案：

1） 前期绑定：在编译时刻，根据 a 和 pa 的静态类型来决定 f 属于哪一个类。由于 a 和 pa 的静态类型分别是 A&和 A*，所以确定 f 是 A::f。

2） 后期绑定：在运行时刻，根据 a 和 pa 实际引用和指向的对象类型（动态类型）来确定 f 属于哪一个类。在 func1(a)和 func2(&a)的调用中，f 是 A::f；在 func1(b)和 func2(&b)的调用中，f 是 B::f。

C++是一个注重程序效率的语言，而采用后期绑定的程序效率是不高的，因此，C++中默认的绑定方式是前期绑定，如果需要后期绑定，必须由程序员在程序中显式指出。

8.5.3 虚函数

为了对上述的 func1 和 func2 中的 f 调用进行后期绑定，必须在类 A 的定义中把 f 声明为虚函数(virtual)：

```
class A
{ …
  public:
   virtual f();
};
```

这样，func1 和 func2 中的 a.f()和 pa->f()将在运行时刻根据 a 和 pa 实际引用和指向的对象类型来确定 f 属于哪一个类。

一旦在基类中指定某成员函数为虚函数，那么，不管在派生类中是否给出 virtual 声明，派生类（派生类的派生类,…）中对其重定义的成员函数均为虚函数。

限制：

1） 类的成员函数才可以是虚函数。

2） 静态成员函数不能是虚函数。

3） 内联成员函数不能是虚函数。

4） 构造函数不能是虚函数。

5） 析构函数可以（往往）是虚函数。

例：

```
class A
{ public:
    A() { f();}
```

```
    virtual f();
    g();
    h() { f(); g(); }
};
class B: public A
{ public:
    f();
    g();
};
…
B b; //调用 b.B::B(), b.A::A(), b.A::f
A *p=&b;
p->f(); //调用 b.B::f
p->g(); //调用 b.A::g
p->h(); //调用 b.A::h, b.B::f, b.A::g
```

8.5.4 纯虚函数和抽象类

1）纯虚函数

只给出了函数声明没给出实现的虚成员函数称为纯虚函数，声明时在函数原型的后面加上 =0，例如：

```
virtual int f()=0;
```

2）抽象类

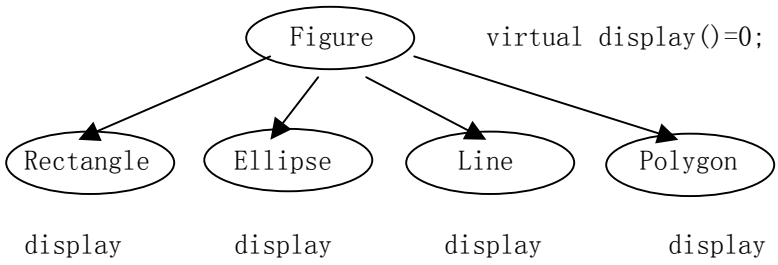
包含纯虚函数的类称为抽象类，抽象类不能用于创建对象。

例：

```
class A //抽象类
{ …
  public:
   virtual int f()=0; //纯虚函数
};
......
A a; //Error
```

抽象类的作用在于为派生类提供一个框架：派生类应提供抽象基类的所有成员函数的实现。

3）抽象类的使用

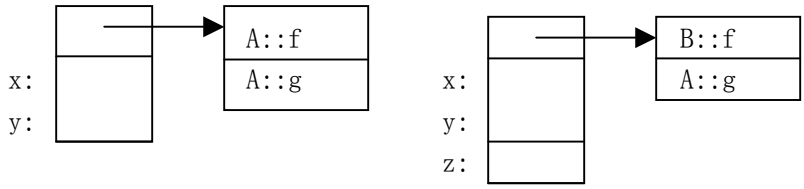



```
Figure *a[100];
a[0] = new Rectangle();
a[1] = new Ellipse();
a[2] = new Line();
…
for (int i=0; i<num_of_figures; i++) a[i]->display();
```

8.5.5 虚函数后期绑定的实现

对于每一个类，如果有虚函数（包括从基类继承来的），则编译器将会为其创建一个虚函数表(vtable)，表中记录了类中所有虚函数的入口地址。当创建一个对象时，在所创建对象的内存空间中有一个指针指向该对象所属类的虚函数表，例如：

```
class A
{   int x,y;
  public:
    virtual f();
    virtual g();
    h();
};
class B: public A
{  int z;
  public:
   f();
   h();
};
…
A a;
B b;


a          A_vtable                  b          B_vtable
```



当通过引用或指针访问对象的虚成员函数时，将利用虚函数表来后期绑定实际调用的函数，例如：

```
A *p;
…
p->f(); // 编译成 (*(*p))(p);
p->g(); // 编译成 (*((char *)*p+4))(p); 假设一个
        // 函数地址占 4 个字节
```

当通过引用或指针访问对象的非虚成员函数和不通过引用或指针来访问对象的成员函数时，不用虚函数表，采用前期绑定，例如：

```
p->h();  //编译成 A::h(p)

a.f();   //编译成 A::f(&a)
a.g();   //编译成 A::g(&a)
a.h();   //编译成 A::h(&a)

b.f();   //编译成 B::f(&b)
b.g();   //编译成 A::g(&b)
b.h();   //编译成 B::h(&b)
```

8.6 类库

MFC(Microsoft Foundation Class Library)。

8.7 *行为继承与实现继承

继承机制是面向对象程序设计中的一个重要特色，采用继承机制带来的明显好处是使得程序的代码量大大减少，并且为实现软件复用提供了一种途径。大多数面向对象程序设计语言（包括C++）提供的是一种用于纯代码复用的实现继承机制，而没有对另一种更深层次的体现设计重用的行为继承(子类型机制)给予足够的支持。不加限制的使用实现继承将使得子类与父类会有完全不同的外部行为，这不仅使得程序难以设计和理解，而且使得程序难以维护与复用。

下面将讨论如何对C++中的继承机制的使用加以限制，使得基类与派生类之间满足行为继承关系。

8.7.1 类型与类

类型刻划了一组值以及可施于其上的一组操作。类型是对其成分的行为进行描述，而不考虑具体实现。当一个变量或表达式属于某类型时，我们所关心的是如何使用它们，而不是它们的实现。类型的设计者、实现者和使用者不必由同一个人担当，类型的设计者给出类型的行为描述，类型的实现者和使用者根据类型的行为描述分别来实现和使用类型。把行为和实现分开，不仅使得程序容易设计和理解，而且当程序有错误时，也容易找到出错的原因。

在面向对象程序设计中，类型描述了一组具有相同性质的对象，这里的性质是指从对象外部可观察到的与实现无关的那些属性，即对象的外部行为。对象的外部行为可以通过向它们发送消息来观察，相同类型的对象将以同样的方式来处理某个消息。

在程序设计语言中一般都提供了类型机制，其好处在于为编译程序提供信息，

- 使得其能进行自动类型检查，以发现程序中的一些错误，提高程序的可靠性
- 产生高效代码，

除此之外，类型机制还对程序的设计和理解有很大的帮助。

类是一种实现机制，它给出了类型的实现，一个类型可以由不同的类来实现。在C++中，把类当作类型来用，一个类型对应一个类，类既用于描述行为又给出了实现，即，对象的行为和实现在C++中合二为一，均用类来描述。这样做的好处在于简化程序设计，不足之处是：它要求同类型的对象必须具有相同的实现，从而使得语言的灵活性下降。

8.7.2 行为继承和实现继承

行为继承是与子类型概念相关的、用于描述行为共享的一种软件复用机制。类型与子类型之间是一种行为(语义)上的继承关系——行为继承，通常又称为子类型关系。具有行为继承（子类型）关系的两个类型之间应满足一种替换原则，即：对所有用类型T来定义的程序P，如果用类型S来替换T，P的行为不变，则称S是T的子类型。上述替换原则表明：子类型对象至少要具有基类型对象的所有行为，也就是说，基类型的操作作用于子类型对象将得到相同的效果，当然，子类型对象还可以具有基类型对象所没有的特殊行为。

采用行为继承的一个好处是可以实现软件的增量开发。在软件开发的早期阶段，我们可能只知道某个数据抽象的一部分操作和一部分行为，随着软件开发过程的进展，对原有的数据抽象又有了新的要求，必须增加新的操作，从而产生新的数据抽象。新的数据抽象可看成是由老的数据抽象精化而来，它们分别由两种类型来描述，这两种类型之间存在子类型(行为继承)关系。另外，行为继承还可实现软件的设计复用，往往，复用设计比复用代码更有利于软件开发。

实现继承是基于类继承的、用于描述实现共享的一种代码复用机制。类与子类间是一种实现上的继承关系——实现继承，子类拥有基类的所有实现，并可给出新功能的实现或对基类已有功能的实现进行重定义。在类继承机制中并没有对子类如何重定义基类的实现作出任何限制，这样，就有可能使得子类与父类有不同的行为，如果把类当作类型来用，就会产生问题。因此，不加限制的使用实现继承会使得程序难以开发、理解与维护。

8.7.3 C++中行为继承的实现

一般认为，C++是一种面向对象程序设计语言，实际上，C++只是支持面向对象程序设计，使用C++并不会使得设计者自动采用面向对象思想进行程序设计，设计者必须掌握一定的面向对象思想才能用C++进行面向对象程序设计。

C++用类来实现数据抽象，用类继承机制来支持实现继承。在C++中，类型的行为描述和实现均由类来担当，类型之间的是否存在子类型关系是由相应的类之间是否存在(实现)继承关系来决定的，这就要求类与子类之间应该具有某种行为(语义)联系。

C++不直接支持类型间的行为继承，但可以用实现继承来实现类型间的行为继承。在C++中，类与子类之间是一种实现继承关系，子类中可以对父类的成员函数(方法)进行重定义，这种重定义可以是任意的，有可能在子类中重定义出的成员函数与父类的成员函数具有完全不同的行为，从而使得它们所表示的类型之间不存在子类型关系。因此，为了保证类与子类之间保持子类型关系，必须要对C++的继承机制的使用加以限制。

根据类型与子类型之间应满足的替换原则，子类型应具有父类型的所有行为，并可具有自己特殊的行为，这里的行为可以理解成类型的各个操作之间应满足的一些恒等式。例如，对于一个公司的所有职员，可以用下述的一个C++类Employee来描述：

```
class Employee
{ public:
   Employee(int n=0) { salary = n; };
   void set_salary(int n) { salary = n; };
```

```
    int get_salary() { return salary; };
  private:
    int salary;
}
```

该类每个对象emp的一个行为可表示成下述的恒等式：

```
[emp.set_salary(n)].get_salary() == n
```

其中，符号[obj.func(para)]表示对对象obj进行func操作之后的、具有新状态的对象obj。

公司各部门的经理也是公司的职员，他们除了具有一般职员的所有行为以外，还具有一些特殊行为，如：一个经理可管理若干职员，下面给出了描述经理的C++类Manager的定义：

```
define max 20
class Manager: public Employee
{ public:
    Employee *get_employee(int n) { return group[n]; };
    void set_employee(Employee *e, int n) { group[n] = e; };
  private:
    Employee *group[max];
}
```

上述定义的类Manager是类Employee的子类，显然，每个类Manager的对象mgr也满足上述的恒等式：

```
[mgr.set_salary(n)].get_salary() == n
```

如果经理的工资还可以包括奖金，奖金数额由某算法bonus()计算得到，则在类Manager的定义中可把类Employee的成员函数set_salary重定义为：

```
Manager::set_salary(int n) { Employee::set_salary(n+bonus()); }
```

这样得到的类Manager将不再满足上述的恒等式，这时，

```
[mgr.set_salary(n)].get_salary() == n+bonus()
```

如果在类Manager中不对set_salary重定义，而是给出一个新的成员函数set_bonus其定义如下：

```
Manager::set_bonus() { set_salary(get_salary()+bonus()); }
```

或者，在类Manager中定义一个新的成员变量Bonus，把set_salary重定义成：

```
Manager::set_salary(int n)
```

```
{ Employee::set_salary(n); Bonus = bonus(); }
```

并定义新的成员函数set_bonus为：

```
Manager::set_bonus() { set_salary(get_salary()+Bonus); }
```

则类Manager的对象仍然满足上述的恒等式。

因此，要使得类与子类之间符合行为继承关系，在定义子类时应采取下述原则：

(1)不对父类的成员函数重定义。

(2)如果对父类的成员函数重定义，则重定义时应以相同的方式改变父类的成员变量，并且要保证重定义的函数作用于子类对象时所得到的返回值，应与父类相应函数作用于父类对象所得到的值相同。

另外，为了保证数据的封装性，子类成员函数最好不要直接操作父类的成员变量，而应通过父类的成员函数来进行，这样，当父类的数据表示发生变化不至于影响到子类的实现。

第9章 模板

9.1 多态性

一般含义是，某一论域中的一个元素可以有多种解释。具体到程序语言，则有以下两个含义：

1) 相同的语言结构可以代表不同类型的实体(一名多用)；
2) 相同的语言结构可以对不同类型的实体进行操作（类属）。

多态的作用是提高语言的灵活性、实现高层软件的复用。

面向对象程序设计具有一种独特的多态：一个公共的消息集可以发送到不同种类的对象，从而得到不同的处理。

绑定（联编、定联）(Binding)：静态（前期）与动态（后期，延迟）绑定（Early Binding&Late Binding)。

强类型语言与弱类型语言。

类属性用于实现参数化模块，即，给程序模块加上类型参数，使其能对不同类型的数据实施相同的操作，它是多态的一种形式。

在C++中，类属性主要体现在类属函数和类属类中。

9.2 函数模板

类属函数指一个函数对不同类型的数据完成相同的操作。

1、宏实现

```
#define max(a,b)  ((a)>(b)?(a):(b))
```

不足之处：只能实现简单的功能，并且存在不作类型检查以及重复计算等问题。

2、函数重载

```
int max(int,int);
double max(double,double);
A max(A,A);
```

不足之处：需要定义的重载函数太多，如果定义不全会有问题。

如：

cout << max(‘3’ ,’ 5’); 将输出 53 而非 ‘5’。

3、指针实现

例：编写一个排序(sort)函数，使其能对各种数据进行排序（整型数序列、浮点数序列以及某个类的对象序列等)。

```
void sort(void *base, //需排序的数据首地址
          unsigned int count, //数据元素的个数
          unsigned int element_size, //数据元素的大小
          int (*cmp)(void *, void *) //比较两个数据元素
                                     //大小的函数指针
          )
{ //取第 i 个元素
      (char *)base+i*element_size
  //比较第 i 个和第 j 个元素的大小
      (*cmp)((char *)base+i*element_size,
            (char *)base+j*element_size
            )
  //交换第 i 个和第 j 个元素
     char *p1=(char *)base+i*element_size,
          *p2=(char *)base+j*element_size;
     for (k=0; k<element_size; k++)
     { char temp=p1[k];
       p1[k] = p2[k];
       p2[k] = temp;
     }
}
int int_compare(void *p1, void *p2)
{ if (*(int *)p1 < *(int *)p2)
    return -1;
  else if (*(int *)p1 > *(int *)p2)
    return 1;
  else
    return 0;
}
int double_compare(void *p1, void *p2)
{ if (*(double *)p1 < *(double *)p2)
    return -1;
```

```
  else if (*(double *)p1 > *(double *)p2)
    return 1;
  else
    return 0;
}
int A_compare(void *p1, void *p2)
{ if (*(A *)p1 < *(A *)p2) //类A需重载操作符：<
    return -1;
  else if (*(A *)p1 > *(A *)p2) //类A需重载操作符：>
    return 1;
  else
    return 0;
}
…
int a[100];
sort(a,100,sizeof(int),int_compare);
double b[200];
sort(b,200,sizeof(double),double_compare);
A c[300];
sort(c,300,sizeof(A),A_compare);
```

不足之处：需要定义额外的参数，并且有大量的指针运算，使得实现起来麻烦、可读性差。

4、函数模板

例：上述的排序用函数模板来实现。

```
template <class T>
void sort(T elements[], unsigned int count)
{ //取第i个元素
    elements [i]
  //比较第i个和第j个元素的大小
    elements [i] < elements [j]
  //交换第i个和第j个元素
     T temp=elements [i];
     elements [i] = elements [j];
     elements [j] = temp;
}
......
int a[100];
```

```
sort(a,100);
double b[200];
sort(b,200);
A c[300]; //类 A 中需重载操作符：<和=，给出拷贝构造函数
sort(c,300);
```

函数模板定义了一类重载的函数，使用函数模板所定义的函数（模板函数）时，编译系统会自动把函数模板实例化。

模板的参数可以有多个，用逗号分隔它们，如：

```
template <class T1, class T2>
void f(T1 a, T2 b)
{ ......
}
```

模板也可以带普通参数，它们须放在类型参数的后面，调用时需显式实例化，如：

```
template <class T, int size>
void f(T a)
{ T temp[size];
  ......
}
void main()
{ f<int,10>(1);
}
```

有时，需要把函数模板与函数重载结合起来用，例如：

```
template <class T>
T max(T a, T b)
{ return a>b?a:b;
}
…
int x,y,z;
double l,m,n;
z = max(x,y);
l = max(m,n);
```

问题：max(x,m)如何处理？

定义一个 max 的重载函数：

```
double max(int a,double b)
{ return a>b?a:b;
}
```

9.3 类模板

类属类指类定义带有类型参数，一般用类模板实现。

例：定义一个栈类，其元素类型可以变化。

```
template <class T>
class Stack
{   T buffer[100];
  public:
    void push(T x);
    T pop();
};
template <class T>
void Stack <T>::push(T x) { …  }

template <class T>
T Stack <T>::pop() { … }
……
Stack <int> st1;
Stack <double> st2;
```

类模板定义了若干个类，类模板的实例化是显式的。

类模板的参数可以有多个，其中包括普通参数，用逗号分隔它们。并且普通参数须放在类型参数的后面。

例：定义不同大小的栈模板

```
template <class T, int size>
class Stack
{   T buffer[size];
  public:
    void push(T x);
    T pop();
};
```

```
template <class T,int size>
void Stack <T,size>::push(T x) { … }

template <class T, int size>
T Stack <T,size>::pop() { … }
……
Stack <int,100> st1;
Stack <double,200> st2;
```

注意：1）类模板不能嵌套（局部类模板）。

2）类模板中的静态成员仅属于实例化后的类（模板类），不同实例之间不存在共享。

9.4 模板是一种代码复用机制

模板提供了代码复用。在使用模板时首先要实例化，即生成一个具体的函数或类。函数模板的实例化是隐式实现的，即由编译系统根据对具体模板函数（实例化后的函数）的调用来进行相应的实例化，而类模板的实例化是显式进行的，在创建对象时由程序指定。

一个模板有很多实例，是否实例化模板的某个实例由使用点来决定，如果未使用到一个模板的某个实例，则编译系统不会生成相应实例的代码。

在 C++中，由于模块是分别编译的，如果在模块 A 中要使用模块 B 中定义的一个模板的某个实例，而在模块 B 中未使用这个实例，则模块 A 无法使用这个实例，除非在模块 A 中也定义了相应的模板。因此模板是基于源代码复用，而不是目标代码复用。

例：

```
// file1.h
template <class T>
class S
{  T a;
  public:
   void f();
};

// file1.cpp
#include "file1.h"
template <class T>
void S<T>::f()
{ …
```

```
}

template <class T>
T max(T x, T y)
{ return x>y?x:y;
}

void main()
{ int a,b;
  float m,n;
  max(a,b);
  max(m,n);
  S<int> x;
  x.f();
}

// file2.cpp
#include "file1.h"
extern double max(double,double);

void sub()
{ max(1.1,2.2); //Error, no appropriate instance
  S<float> x;
  x.f(); //Error, corresponding instance has no appropriate implementation
}
```

第10章 输入/输出（I/O）

10.1 概述

输入/输出(I/O)是程序的一个重要组成部分，也是一个不容易规范化的语言机制，它要依赖于不同的操作系统，是平台有关的。

输入/输出不是C++语言的成分，而是由具体的实现（编译器）作为函数库或类库来提供。

C++的设计者对I/O提出了一种方案，虽然它不属于C++的标准范畴，但大多数的实现也都实现了这个方案，因此，它成了默认的标准。

在C++中，输入/输出操作是基于一种字节流来实现的。在进行输入操作时，数据逐个字节地从外部流入到内部（计算机中）；在进行输出操作时，数据逐个字节地从内部流出到外部。

为了方便使用，在C++的函数库和类库中，除了提供基于字节的I/O操作外，还提供了C++基本数据类型数据的I/O操作。另外，也可以对类库中I/O流类的一些操作进行重载，使其能对自定义类的对象进行I/O操作。

10.2 基于函数库的I/O（面向过程）

用C++以面向过程风范进行程序设计时，可使用C++函数库中的函数实现输入/输出。这些函数分别用于控制台、文件和字符串变量的I/O，下面分别介绍它们。

10.2.1 控制台I/O

1、输出

```
#include <stdio.h>
putchar(int ch); //ch 中的字符输出到显示器
puts(const char *p); //p 所指向的字符串输出到显示器

printf(const char *format [,<参数表>]); //提供基本类型数据的输出
```

其中，format指向一个格式字符串，该字符串包含两类字符：普通显示字符和控制字符。普通字符将显示在显示器上，控制字符用于控制<参数表>中的数据的显示格式。<参数表>中的参数为表达式。

例：

```
int i=1,j=2;
printf("i=%d, j=%d\n",i,j);
```

结果为：

```
i=1, j=2
```

其中格式字符串中的两个"%d"用于控制把参数表中的变量 i 和 j 按照十进制整型数输出。

控制字符以%开始，常用的控制符号有：

%c 字符
%d 十进制整型数
%u 无符号十进制整型数
%o 八进制整型数
%x 十六进制整型数
%s 字符串
%f 浮点数，包括 float 和 double，格式为：dddd.dddd
%e 浮点数，包括 float 和 double，格式为：[-]*d.dddd* e[+/-]*ddd*

2、输入

```
#include <stdio.h>
int getchar(); //从键盘输入一个字符返回。
char *gets(const char *p); //从键盘输入一个字符串放入 p
                           // 所指向的内存空间。

scanf(const char *format [,<参数表>]);
```

其中，format 指向一个格式字符串，该字符串包含两类字符：普通显示字符和控制字符。普通字符用于与输入字符进行匹配，控制字符用于控制<参数表>中的数据的输入格式。<参数表>中的参数为变量的地址。

例：

```
int i,j;
scanf("i=%d,j=%d",&i,&j);
```

输入为：

```
i=10,j=20
```

控制字符与 printf 类似。

另外，有些 C++编译器还提供了一些特殊的函数，如：

```
#include <conio.h>
int getch();//不带缓冲的输入，不回显
int getche();//不带缓冲的输入，回显
```

10.2.2 文件 I/O

1、输出

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode ); //打开文件
```

其中，mode 可以是：

"w"：打开一个文件用于写操作，如果文件已存在，则首先清空之。

"a"：打开一个文件用于添加（从文件末尾）操作。

另外，在 w 或 a 的后面还可以加上 t 或 b，以区别文本或二进制方式，默认方式由全局变量_fmode 指定，通常为文本（t）方式。在文本方式下，'\n'字符自动转换成'\r'和'\n'两个字符。

```
int fputc( int c, FILE *stream ); //输出一个字符
int fputs( const char *string, FILE *stream ); //输出一个字符串
int fprintf( FILE *stream, const char *format [, argument ]...);
        //输出基本类型数据
size_t fwrite( const void *buffer, size_t size, size_t count,
            FILE *stream ); //按字节流输出数据

fclose( FILE *stream); //关闭文件
```

2、输入

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
```

其中，mode 可以是：

"r"：打开一个文件用于读操作。

另外，在 r 的后面还可以加上 t 或 b，以区别文本或二进制文件。在文本方式下，'\r'和'\n'两个字符自动转换成'\n'一个字符，0x1A 表示文件结束。

```
int fgetc(FILE *stream );
char *fgets(char *string,int n, FILE *stream );
int fscanf( FILE *stream, const char *format [, argument ]...);
size_t fread( const void *buffer, size_t size, size_t count,
            FILE *stream );
feof(FILE *stream); //判断文件结束
fclose( FILE *stream);
```

3、输入/输出

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
```

其中，mode 可以是：

"r+"：打开一个文件用于读/写操作，文件必须存在。

"w+"：打开一个空文件用于读/写操作，如果文件已存在，则首先清空之。

"a+"：打开一个文件用于读/添加操作。

另外，在 r+、w+或 a+的后面还可以加上 t 或 b，以区别文本或二进制文件。

4、随机输入/输出

每个打开的文件都有一个位置指针，指向当前读写位置，每读或写一个字符，位置指针都会自动往后移动一个位置。

```
int fseek( FILE *stream, long offset, int origin );
        //显式指定位置指针的位置
```

其中，origin 指出移动的参考位置，可以是 SEEK_CUR（当前），SEEK_END（末尾）或 SEEK_SET（起始）；offset 为偏移量。

```
long ftell( FILE *stream ); //返回位置指针的位置。
```

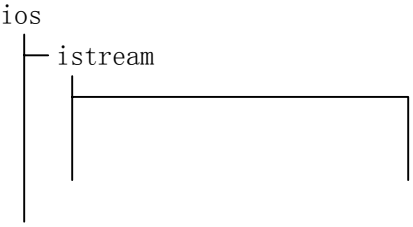
10.2.3 字符串变量 I/O

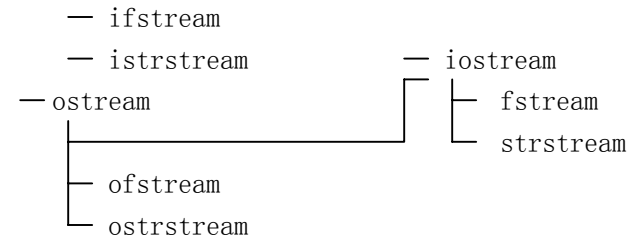
```
int sprintf( char *buffer, const char *format [, argument] ... );
int sscanf( const char *buffer,
          const char *format [, argument ] ... );
```

10.3 基于类库的 I/O（面向对象）

10.3.1 基本的 I/O 流类及其操作

I/O 类库提供的输入/输出操作是由一些类来实现的，其中 istream(输入流)和 ostream(输出流)是两个基本的类，由它们派生出其它的类，如：iostream(输入/输出流)等，这些类是在头文件 iostream.h 中声明的。I/O 类库中的主要的类以及它们之间的关系如下图所示：





I/O 流类提供了两个基本操作：>>(抽取，由 istream 类提供)和<<(插入，由 ostream 类提供)用于对基本数据类型的数据进行输入/输出。

需要输入时，可创建一个 istream 类（或其派生类）的对象 o，然后调用对象 o 的操作>>进行输入：

```
o >> x; //x 是一个变量
```

需要输出时，可创建一个 ostream 类（或其派生类）的对象 o，然后调用对象 o 的操作<<进行输出：

```
o << x; //x 是一个表达式
```

I/O 流类库提供了三类输入/输出操作：

- 控制台 I/O：从键盘输入/输出到显示器或打印机。
- 文件 I/O：从文件输入/输出到文件。
- 字符串 I/O：从字符串变量输入/输出到字符串变量。

下面分别介绍这些操作。

10.3.2 控制台 I/O

在 I/O 流库中预定义了四个流对象，用于进行控制台的输入/输出：

- cin： istream 类的对象，对应键盘。
- cout：ostream 类的对象，对应显示器。
- cerr：ostream 类的对象，对应显示器。
- clog：ostream 类的对象，对应打印机。

进行控制台输入/输出，程序中需要包含：

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

1、输出

任何基本数据类型的数据和指针都可以通过 cout、cerr、clog 对象和操作符<<进行输出。

例：

```
int x;
float f;
char ch;
int *p=&x;
......
cout << x ;
cout << f;
cout << ch;
cout << “hello”;
cout << p;
```

或

```
cout << x << f << ch << “hello”<< p;
```

特例：

```
char *p= “abcd”;
cout << p; //输出字符串“abcd”而不是字符串“abcd”的首地址。
cout << (void *)p; //输出字符串“abcd”的首地址，即变量 p 的值。
```

为了控制输出行为，可通过输出相应控制字符来实现，如’\n’表示换行等。控制输出行为也可以通过输出操纵符(manipulator)来实现，如：

```
cout << hex << x << endl;
```

其中的 hex,endl 就是操纵符，上述语句表示以十六进制输出 x 的值，然后换行。

由于输出操作往往是带缓冲的，输出的内容不一定立即能在输出设备上出现，如果要立即输出缓冲中的内容，可以用操纵符 flush 来实现：

```
cout << i << j << flush;
```

注意：在用<<进行输出时，需要注意运算符的优先级，如：

```
int i=10,j=20;
cout << (i>j)?i:j << ‘\n’;
```

结果为：0（为什么？）

另外，利用 cout 输出时，也可以用 ostream 类提供的基于字节流的操作来进行，如：

```
cout.put(char ch);
cout.write(const char *str,int length);
```

2、输入

任何基本数据类型的数据都可以通过 cin 对象和操作符>>进行输入。

例：

```
int x;
double y;
char str[100];
cin >> x; cin >> y; cin >> str;
```

或

```
cin >> x >> y >> str;
```

输入时也可以通过一些操纵符来控制输入的行为，如：

```
cin >> setw(99) >> str;
```

除了操作符>>外，还可以使用 cin 的基于字节流的操作来进行输入，如：

```
cin.get(char &ch);
cin.getline(char *buf, int length, char delim=' \n' );
```

3、操作符<<和>>的重载

为了能对自定义类的对象进行输入/输出，需要对<<和>>进行重载。操作符<<和>>只能作为全局(友元)函数重载（why），下面以操作符<<的重载为例，介绍如何重载<<和>>。

```
class A
{   int x,y;
  public:
    …
    friend ostream& operator << (ostream&,A&);
};
ostream& operator << (ostream& out,A& a)
{   out << a.x <<  ',' << a.y << endl;
    return out;
}
…
A a,b;
cout << a << b;
```

问题：如果 B 是 A 的派生类：

```
class B: public A
```

```
{   double z;
    …
};
…
B b;
cout << b; //只显示 b.x 和 b.y，  而没有显示 b.z
```

由于没有对 B 重载操作符<<，B 类对象的输出按 A 类对象进行（注意：这不是继承！是什么？）。

解决：

方案 1

```
class B: public A
{   double z;
  public:
    …
    friend ostream& operator << (ostream&,B&);
};
ostream& operator << (ostream& out,B& b)
{   out << b.x <<  ‘,’  << b.y << ‘,’  << b.z << endl;
    return out;
}
```

虽然该方案解决了上面的问题，但解决不了下面的问题：

```
A *p;
…
cout << *p;
```

方案 2

```
class A
{   int x,y;
  public:
    …
    virtual void display(ostream& out)
    { out << x <<  ‘,’  << y << endl;
    }
};
ostream& operator << (ostream& out,A& a)
{   a.display(out);
    return out;
```

```
}

class B: public A
{   double z;
  public:
    …
    void display(ostream& out)
    { A::display();
      out <<  ‘,’ << z << endl;
    }
}
```

10.3.3 文件 I/O

对外部文件进行输入/输出，程序中需要：

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

1、输出

首先创建一个 ofstream 类（在 I/O 流库中定义的，是 ostream 类的派生类）的对象：

```
ofstream out_file(<文件名>,<打开方式>,<共享方式>);//直接
```

或

```
ofstream out_file; //间接
out_file.open(<文件名>,<打开方式>,<共享方式>);
```

其中，<文件名>为对象 out_file 对应的外部文件名，<打开方式>可以是：ios::out，ios::app 等，<共享方式>可以是 filebuf::sh_none，filebuf::sh_read 等。

由于种种原因，打开文件操作可能失败。因此，打开文件时应判断打开是否成功，只有文件打开成功后才能对文件进行操作。判断文件是否成功打开可以采用以下方式：

```
if (!out_file)
{ //失败
}
```

或

```
if (!out_file.is_open())
{ //失败
}
```

文件成功打开后，可以使用操作符<<或 ofstream 类的其它操作进行输出，如：

```
int x;
```

```
double y;
.......
out_file << x << y << endl;
out_file.put('A');
out_file.write("ABCDEFG",7);
```

最后，使用 close 关闭文件：

```
out_file.close();
```

2、输入

首先创建一个 ifstream 类（在 I/O 流库中定义的，是 istream 类的派生类）的对象：

```
ifstream in_file(<文件名>,<打开方式>,<共享方式>);
```

或

```
ifstream in_file;
in_file.open(<文件名>,<打开方式>,<共享方式>);
```

其中，<文件名>为对象 in_file 对应的外部文件名，<打开方式>可以是：ios::in 等，<共享方式>可以是 filebuf::sh_none，filebuf::sh_read 等。

必须对文件打开是否成功进行判断，判断方式同输出文件。

然后，使用操作符>>或 ifstream 类的其它操作进行输入，如：

```
in_file >> x >> y;
in_file.get(ch);
in_file.getline(buf, length, '\n');
in_file.read(buf,length);
```

在文件输入操作过程中，常常需要判断文件是否已结束（内容被读完），可以调用：

```
in_file.eof()
```

它返回 0 表示未结束，非 0 表示结束。

最后，使用 close 关闭文件：

```
in_file.close();
```

注意：从文件中读数据时必须知道所读文件中的数据格式！包括数据的类型和数据的存

贮方式。存贮方式有文本方式(text)和二进制方式(binary)。

文本方式存贮的文件和二进制方式存贮的文件的主要区别是：

文本文件中只包含可显示字符(ASCII 编码为：0x20~0x7F)和若干个控制字符：0x0D('\r')、0x0A ('\n')、0x09('\t')以及 0x1A(文件结束符)，一般用于存贮不带格式信息（回车、换行以及制表（横向跳格）除外）的文本、源程序以及文本数据等。

而二进制文件中包含任意的二进制字节，一般用于存贮带格式信息文本、目标代码程序以及二进制数据等。

例、对于一个整型数 1234567，可以用两种方式保存到文件中：

a) 依次把 1,2,3,4,5,6,7 的 ASCII 码(7 个字节)写入文件（文本文件），文件中包含 7 个字节。

b) 把 1234567 的补码（内部表示）0x0012D687（4 个字节）写入文件（二进制文件），文件中包含 4 个字节。

当从上述文件中读取数据时，首先要知道读取的数据种类（整型），其次要知道该数据是以什么方式存贮的（text 或 bibary），否则无法读入数据。如：

```
int i=1234567,j;
out_file << i;            //产生格式 a)的数据文件
out_file.write(&i,4);     //产生格式 b)的数据文件
.......
in_file >> j; //能正确读入格式 a)中的数据，不能正确读入格式 b)
              //中的数据。
in_file.read(&j,4); //能正确读入格式 b)中的数据，不能正确读入
                    //格式 a)中的数据（同一种环境）。
```

在打开文件时，除了要指定文件的读写方式(ios::out,ios::app,ios::in，等)外，还可以指定文件是 text 还是 binary，默认的是 text。

在打开方式中，可以把 ios::binary 通过运算符'|'(按位或)与 ios::out、ios::app 或 ios::in 结合，指定二进制文件。如：

```
ios::in|ios::binary          //打开一个二进制文件读
ios::out|ios::binary     //打开一个二进制文件写
```

如果以 text 方式打开文件（读或写），在读操作中，遇见连续的 0x0D 和 0x0A 两个字符时，系统会把它们转换成一个字符 0x0A('\n')读入；在写操作中，输出字符 0x0A('\n')时，系统把它转换成两个字符 0x0D 和 0x0A 写入文件。另外，在读操作中遇见字符 0x1A，系统认为是文件结束。因此，以文本方式打开一个二进制文件读会造成数据的丢失。

例子：分别以文本和二进制方式输出和输入学生记录。

```
#include <fstream.h>
const int STUDENT_NUM=2;
enum SEX {MALE, FEMALE};
struct STUDENT
{ int no;
  char name[20];
  int age;
  SEX sex;
  char addr[80];
} students[STUDENT_NUM]={{1,"wang gang",18,MALE,"ABCDEFGH"},
                         {2,"li ming",19,FEMALE,"abcdefgh"}
                        };

void main()
{ int i;

  ofstream out_file;

  //以文本文件格式输出
  out_file.open("d:\\students.txt",ios::out);
  if (!out_file.is_open()) return;

  for (i=0; i< STUDENT_NUM; i++)
  { out_file << students[i].no << ","
             << "\"" << students[i].name << "\","
             << students[i].age << ",";
    if (students[i].sex == MALE)
      out_file << "m,";
    else
      out_file << "f,";
    out_file << "\"" << students[i].addr << "\"\n";
  }
  out_file.close();

  //以二进制文件格式输出
  out_file.open("d:\\students.dat",ios::out|ios::binary);
  if (!out_file.is_open()) return;

  for (i=0; i< STUDENT_NUM; i++)
```

```
  out_file.write((char *)&students[i],sizeof(STUDENT));
out_file.close();

//以文本文件格式输入
ifstream in_file;
in_file.open("d:\\students.txt",ios::in);
if (!in_file.is_open()) return;
in_file.unsetf(ios::skipws); //不跳过空格、制表以及回车符

int no,j;
char ch;

for (i=0,in_file>>no; !in_file.eof(); i++,in_file>>no)
{ students[i].no = no;
  in_file >> ch; //读入：,

  in_file >> ch; //读入："
  for (j=0,in_file>>ch; ch != '"'; j++,in_file>>ch)
    students[i].name[j] = ch;
  students[i].name[j] = '\0';
  in_file >> ch; //读入：,

  in_file >> students[i].age;
  in_file >> ch; //读入：,

  in_file >> ch;
  if (ch == 'm')
    students[i].sex = MALE;
  else
    students[i].sex = FEMALE;
  in_file >> ch; //读入：,

  in_file >> ch; //读入："
  for (j=0,in_file>>ch; ch != '"'; j++,in_file>>ch)
    students[i].addr[j] = ch;
  students[i].addr[j] = '\0';

  in_file >> ch; //读入：\n
}
in_file.close();
```

```
  //以二进制方式输入
  in_file.open("d:\\students.dat",ios::in|ios::binary);
  if (!in_file.is_open()) return;

  for (i=0,in_file.read((char *)&students[i],sizeof(STUDENT));
       !in_file.eof();
       i++,in_file.read((char *)&students[i],sizeof(STUDENT)))
    ;
  in_file.close();

}
```

3、输入/输出

如果打开一个文件，既能读入，也能输出，则需要创建一个 fstream 类的对象，fstream 类是从 iostream 类派生的。

例：fstream file("d:\\test.txt",ios::in|ios::out);

4、随机存取文件

默认情况下，I/O 流库中的类所提供的输入/输出操作是顺序进行的，即，在读入时，要读文件中的第 n 个数据，必须先把前 n-1 个数据读入；在写出时，要写文件中的第 n 个数据，必须先把前 n-1 个数据写出。

每个文件都有一个内部文件位置指针，读/写时，每读/写一个字节，这个指针会自动往后走一个字节位置。

为了能够随机读写文件，必须显示地指出读写的位置。读写位置可通过设置文件内部指针来指定。

对于输入文件，可用下面的操作来指定文件内部指针位置：

```
istream& istream::seekg(<位置>);
istream& istream::seekg(<偏移量>,<参照位置>);
streampos istream::tellg();
```

对于输出文件，可用下面的操作来指定文件内部指针位置：

```
ostream& ostream::seekp(<位置>);
ostream& ostream::seekp(<偏移量>,<参照位置>);
streampos ostream::tellp();
```

<参照位置>: ios::beg, ios::cur, ios::end

5、重载

同控制台 I/O。

10.3.4 字符串 I/O

用于从字符串变量输入和向字符串变量输出。　　需要：

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

1、输出

首先创建一个 ostrstream 类（在 I/O 流库中定义的，是 ostream 类的派生类）的对象：

```
ostrstream str_buf;
```

或

```
ostrstream str_buf(char *p, int length, <方式>);
```

其中，p 为一个缓冲指针，length 为缓冲大小，<方式>可以是 ios::out，ios::app 等。如果 str_buf 采用默认构造，则采用可动态扩充的内部缓冲。

使用操作符<<或 ostrstream 类的其它操作进行输出，如：

```
str_buf << x << y << endl;
```

使用 str 操作获取 str_buf 中字符串缓冲头指针：

```
char *p=str_buf.str();
```

2、输入

首先创建一个 istrstream 类（在 I/O 流库中定义的，是 istream 类的派生类）的对象：

```
istrstream str_buf(char *p);
```

或

```
istrstream str_buf(char *p, int length);
```

其中，p 为一个缓冲指针，length 为缓冲大小。如果 str_buf 的构造没有给出 length，则认为 p 以‘\0’结束。

使用操作符>>或 istrstream 类的其它操作进行输入，如：

```
str_buf >> x >> y;
```

可以使用 str 操作获取 str_buf 中字符串缓冲头指针：

```
char *p=str_buf.str();
```

3、重载

同控制台 I/O。

第11章 异常处理

11.1 异常的概念

程序的错误包括：语法错误，逻辑错误和运行异常：

● 语法错误是指程序的书写不符合语言的语法规则，由编译程序发现。

● 逻辑错误或语义错误是指程序设计不当造成程序没有完成预期的功能，通过测试发现。

● 运行异常是指由程序运行环境问题造成的程序异常终止，如：内存空间不足、打开不存在的文件、程序执行了除以 0 的指令，等等。

下面就运行异常的处理进行介绍。

在程序运行环境正常的情况下，导致运行异常的错误是不会出现的，这类错误是可以预料的，但是无法避免。为了保证程序的鲁棒性(Robustness)，必须在程序中对它们进行预见性处理。如：

```
void f(char *str)
{ ifstream file(str);
  if (file.fail())
  { … //异常处理
  }
  int x;
  file >> x;
  …
}
```

上述的异常处理是在发现错误的地方（函数）就地处理，有时就地处理往往不合适，而需要由调用的函数来处理。例如：在上述的函数 f 中用到的文件名 str，可能是在调用 f 的函数中，通过某种方式（如：用户输入）得到的，当函数 f 发现文件不存在，应该让调用者重新获得文件名，然后重新调用 f，即，发现异常的函数和处理异常的函数可以不是同一个函数。

语言中如果没有一种专门的异常处理机制，要想实现上述的异常处理思想有时是比较麻烦的，并且，也会使得程序结构非常不清楚。

例如：被调函数通过返回不同的值来告诉调用函数有关被调函数的执行情况(正常、异常)；调用函数根据被调函数的返回值作出不同的处理（继续执行、处理异常或返回到调用函数的调用函数进行异常处理）。

上述做法，首先，会使得程序对正常执行过程的描述显得十分不清楚。其次，对一个原来没有返回值的被调函数，为了反映调用是否正常，往往要把它定义成有返回值，用于返回它的执行情况。另外，对一个正常执行后必须返回一个值的被调函数，调用函数往往无法根据被调函数的返回值来判断调用是否成功，这时，需要采用其它方式来解决。

11.2 C++异常处理机制

C++提供了一种能够清晰描述异常处理过程的机制，其主要思想是：把有可能造成异常的一系列操作（语句或函数调用）构成一个 try 语句块，如果 try 语句块中的某个操作在执行中发现了异常，则通过执行一个 throw 语句抛掷（产生）一个异常对象；抛掷的异常对象将由能够处理这个异常的地方通过 catch 语句块来捕获并处理之。

1 try 语句块

启动异常处理机制。

格式：

```
try
{ <语句序列>
}
```

上述的<语句序列>中可以有函数调用。

例如：

```
try
{ f(filename);
}
```

2 throw 语句

抛掷（产生）异常对象，其后的语句不再继续执行，转向异常处理。

格式：

```
throw <表达式>
```

例如：

```
void f(char *str)
{ ifstream file(str);
  if (file.fail())
```

```
    { throw str; //产生异常对象
    }
    ......
  }
```

3 catch 语句块

捕获异常对象并处理相应的异常，它紧接在某个 try 语句的后面。

格式：

```
  catch (<类型> [<变量>])
  { <语句序列>
  }
```

其中，<类型>用于指出捕获何种异常对象，<类型>与异常对象的类型匹配规则同函数重载；<变量>用于存贮异常对象，它可以缺省，缺省时表明只关心异常对象的类型，而不考虑具体异常对象。

例如：

```
  char filename[100];
  …
  cout << “请输入文件名：” << endl;
  cin >> filename;
  try
  { f(filename);
  }
  catch (char *str)
  { cout << str << “不存在!”<< endl;
    cout << “请重新输入文件名：” << endl;
    cin >> filename;
    f(filename);
  }
```

一个 try 语句块的后面可以跟多个 catch 语句块,用于捕获不同类型的异常进行处理。

例如：

```
void f()
{ ......
  throw 1;
```

```
  ......
  throw 1.0;
  ......
  throw "abcd";
  .......
}
void main()
{ ......
  try
  { f();
  }
  catch (int) //处理 throw 1;
  { <语句序列 1>
  }
  catch (double) //throw 1.0
  { <语句序列 2>
  }
  catch (char *) //throw "abcd"
  { <语句序列 3>
  }
  <非 catch 语句>
  ......
}
```

注意：

1）在 try 的<语句序列>执行中如果没有抛掷(throw)异常对象，则其后的 catch 语句不执行，继续执行 try 以后的非 catch 语句。

2）在 try 的<语句序列>执行中如果抛掷(throw)了异常对象，而其后有能够捕获该异常的 catch 语句，则执行该 catch 语句中的〈语句序列〉，然后继续执行 try 以后的非 catch 语句。

3）在 try 的<语句序列>执行中如果抛掷(throw)了异常对象，而其后没有能够捕获该异常的 catch 语句，则由上一层中的 try 语句的 catch 来捕获。

4)如果执行的语句不在 try 的<语句序列>中,则抛掷的异常不会被程序中的 catch 捕获。

11.3 异常处理的嵌套

例子：f→g→h

```
h()
{ …
  throw 1; //由 g 捕获并处理
  …
  throw “abcd”; //由 f 捕获并处理
}
g()
{ try
  { h();
  }
  catch (int)
  { …
  }
}
f()
{ try
  { g();
  }
  catch (int)
  { …
  }
  catch (char *)
  { …
  }
}
```

如果抛掷的异常对象在调用链上没有给出捕获，则调系统的 abort 作标准异常处理。

11.4 例子

```
#include <iostream.h>
int h(int x, int y)
{ if (y == 0) throw 0;
  return x/y;
```

```
}

void f()
{ int a,b;
  try
  { cin >> a >> b;
    cout << "a/b is " << h(a,b);
  }
  catch(int)
  { cout << "The divisor cannot be zero" << endl;
    cout << "Please input once again:" << endl;
    cin >> a >> b;
    cout << "a/b is " << h(a,b);
  }
}

void main()
{ try
  { f();
  }
  catch (int)
  { cout << "In main's exception handling"<< endl;
  }
}
```

①输入 1　2　② 输入 3 0 4 2　③ 输入 3 0 4 0

第12章 类库（MFC）

12.1 软件开发环境

有很长一段时间，软件工作者所做的大量工作都是为了使得他人能够很方便地使用计算机，而自己却没有得到任何的获益，仍然在一个难用的计算机环境中工作。为了解决软件工作者自己使用计算机难的问题，软件开发环境应运而生。

12.1.1 定义

支持一定的软件开发方法或按一定的软件开发模型组织起来的一组相关的软件工具的集合。

12.1.2 软件开发环境的构成

1） 软件信息数据库（环境的核心）。
2） 交互式的人机界面。
3） 工具集，包括：分析、设计、实现、测试/调试、维护、质量保证、配置管理、语言等。其中的语言不仅仅指编程语言，每个工具都应该有相应的语言。

12.1.3 软件开发环境的种类

1） 工具箱式，如：UNIX 环境。
2） 面向语言的程序设计环境，如 Ada 环境（APSE）、Smalltalk 环境。
3） 支持整个软件生存周期的软件工程环境（CASE），包括支持一种开发方法和支持多种开发方法。
4） 软件信息数据库环境。

目前，软件开发环境发展趋势是：集成化、智能化和支持重用。

12.1.4 面向对象程序设计环境

面向对象程序设计环境具有以下特点：
1） 语言与环境融为一体。
2） 开放性（可扩充性）。
3） 集成性（对象）。

面向对象程序设计环境的核心是类库。类库和函数库（子程序库）都是支持代码复用的重要手段，它们的区别在于：

1） 子程序库中的一个子程序仅完成一个独立的功能；而类库中的一个类具有一组功能。

2） 子程序库中的各子程序彼此独立，组织比较松散；而类库中的各个类之间存在一定的联系，组织紧密。

3） 子程序通过参数接受外界数据，信息量有限；而类（对象）自己有局部数据。

4） 类具有动态绑定特性。

5） 当需要修改功能时，子程序必须重新定义；而类可以通过继承来实现。

下一节简要介绍微软公司的 MFC 类库。

12.2 MFC(Microsoft Foundation Class library)类库

MFC 中大部分的类主要用于实现 Windows 应用程序的功能，它们封装了 Windows 应用程序中一些元素（如：窗口）的基本功能，因此，在介绍 MFC 主要内容之前，首先介绍一下 Windows 应用程序的基本构成。

12.2.1 Windows 应用程序的基本构成

1、用户界面

- 单文档
- 多文档
- 对话框

2、程序结构

Windows 应用程序的结构是基于消息驱动模型，程序的任何一个动作都是在接收到一条消息后发生的，如：WM_KEYDOWN/WM_KEYUP，WM_CHAR, WM_LBUTTONDOWN/WM_LBUTTONUP，WM_PAINT, WM_COMMAND 等，每条消息都可以带有参数（wParam，lParam）。

大部分的消息都关联到某个窗口，每个窗口都有一个消息处理过程（函数），属于某个窗口的消息都将由相应的消息处理过程来进行处理。

每个 Windows 应用程序都有一个消息队列，Windows 系统会把属于各个应用程序的消息放入各自的消息队列，应用程序不断地从自己的消息队列中获取消息并把它们发送到相应的窗口处理过程。这个循环（取消息-处理消息）一直到用户以某种方式（如：关闭应用程序的主窗口）结束程序而终止。下面给出了 Windows 应用程序的一个框架：

```
LRESULT CALLBACK WindowProc(HWND hWnd,UINT message,WPARAM wParam,LPARAM lParam);

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{   //注册窗口类
    RegisterClass(…, WindowProc,…,"my_window_class");
    …
    //创建窗口
    CreateWindow("my_window_class",…);
    …
    //消息循环，直到接收到 WM_QUIT 消息
    while (GetMessage(&msg, NULL, 0, 0))
    {   TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return msg.wParam;
}

LRESULT CALLBACK WindowProc(HWND hWnd,UINT message,WPARAM wParam,LPARAM lParam)
{ switch (message)
  { case WM_CHAR:
        …
    case WM_COMMAND:
        switch (wParam)
        {
        }
        …
    case WM_PAINT:
        …
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

. . . . . .

Windows 应用程序用到的大多数系统定义的类型和函数都在 windows.h 中声明。

12.2.2 MFC 主要类介绍

1、MFC 的类层次结构

2、CWnd

提供了一般窗口的基本功能，有许多的功能声明为虚函数，由派生类提供进一步的实现。

3、CFrameWnd

提供了框架窗口的基本功能，特别是管理子窗口的功能。单文档应用的主窗口为 CFrameWnd 派生类的对象。

4、CMDIFrameWnd

提供了多文档应用的主窗口的基本功能，从 CFrameWnd 派生而来。

5、CMDIChildWnd

提供了多文档应用的子窗口的基本功能，从 CFrameWnd 派生而来。

6、CView(视)

提供用于显示应用程序输出结果窗口的功能。其典型的派生类有：CScrollView，CEditView，CFormView，CHtmlView 等。

7、CDocument(文档)

CDocument 类的对象用于存放在 CView 对象中显示的内容。可以在一个 CDocument 对象和多个 CView 对象之间建立联系，使得从一个 CDocument 对象可以找到与之关联的某个 CView 对象；从一个 CView 对象可以找到与之关联的 CDocument 对象。

8、CWinApp

提供 Windows 应用程序的控制功能，如消息循环。每个基于 MFC 的 Windows 应用程序必须有一个 CWinApp 派生类的全局对象，以保证在 WinMain 执行前该对象已创建。在 CWinApp 类中提供了一个成员函数 Run 实现了消息循环；

9、CDocTemplate

用于支持 MFC 文档/视结构，创建三个对象（CFrameWnd,CView 和 CDocument）并建立起它们之间的联系。CDocTemplate 为抽象类，很多功能由其派生类（CSingleDocTemplate 和 CMultiDocTemplate）实现。

10、CSingleDocTemplate

用于单文档应用中的三个对象（CFrameWnd,CView 和 CDocument）的创建。

11、CMultiDocTemplate

用于多文档应用中的三个对象（CFrameWnd,CView 和 CDocument）的创建。

12、CDialog

实现对话框的功能，它必须对应一个对话框模板（用资源编辑器设计）。

```
CMyDlg dlg;
dlg.m_  //设置对话框中的初始内容
…
dlg.DoModal(); //返回值可以为：IDOK 和 IDCANCEL
dlg.m_  //取对话框中的内容
…
```

13、CCommonDialog

系统定义的一些对话框。

- CFileDialog
- CFontDialog
- CColorDialog
- CPageSetupDialog
- CPrintDialog
- CFindReplaceDialog
- COleDialog
- ……

14、CControl

实现对话框中的一些控制的功能。

- CButton
- CEdit
- CListBox

- CToolBarCtrl
- CComboBox
- .....

15、CDC

实现绘图功能。

Windows 应用程序的输出一般需关联到某个窗口，即在某个窗口中输出。输出时，首先要获得相应窗口的输出环境或设备上下文（Device Context,DC），该输出环境包含了输出所需要的各种元素，如：字体、颜色、刷子、窗口在桌面上的位置等等。然后调用 Windows 提供的输出（绘图）函数进行输出，这些函数都需要一个 CDC 类的对象作为它们的参数。

```
CClientDC dc(this); //this 指向某个 CWnd 或其派生类对象
dc.SelectObject(…);
dc.TextOut(x,y,” hello”);
dc.Rectangle(x1,y1,x2,y2);
......
```

16、CFile

实现文件 I/O 功能。

```
CFile file;
file.Open(filename,open_flag);
file.Read(buf,size);
file.Write(buf,size);
file.Flush();
file.Seek(…);
......
file.Close();
```

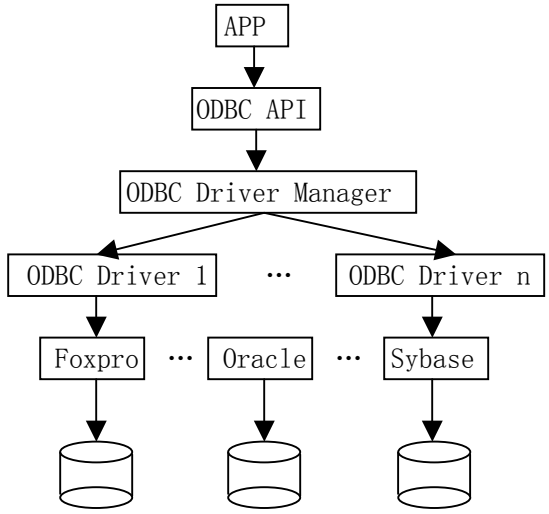
17、CArchive

实现数据的序列化(线性化)（serialization）,即把一个非线性化的结构变成一个线性结构。一个 CArchive 对象包含了一个 CFile 对象，它除了能按 CFile 进行 I/O 外，还提供了对基本数据类型和自定义数据（对象）的 I/O，并重载了操作符<<和>>，如果对自定义的整个对象进行 I/O，对象的类应从 CObject 继承。

18、ODBC: CDatabase,CRecordset

实现以 ODBC 方式使用数据库。

ODBC:



数据源(Data Source)：指定某个具体数据库的名、类型和数据库的位置。
数据源管理器(Data Source Manager)：增加、修改、删除数据源。

```
CDatabase database;
database.Open(<数据源名>,…);
database.Close();

CRecordset table(&database);
table.Open(OpenType,SQL,Option);
table.Close();
```

CRecordset的主要成员函数:IsBOF, IsEOF, IsDeleted, AddNew, Delete, Edit, Update, Move, MoveFirst, MoveLast, MoveNext, MovePrev

上述类均从 CObject 类继承而来，在 MFC 中还有一些类不从 CObject 继承，如：

1） CPoint,CRect,CSize, CString
2） CRuntimeClass

CObject 类有一个成员函数：

```
virtual CRuntimeClass* GetRuntimeClass( ) const;
```

该函数能够得到 CObject 类或其派生类的对象所属的类信息并保存在一个 CRuntimeClass 类的对象中，而 CRuntimeClass 类有一个成员函数：

```
CObject* CreateObject( );
```

该函数能够创建一个在 CRuntimeClass 类的对象中保存的类的一个对象。

例：

```
class A: public CObject
{ .......
};
......
A *p,*q;
…
q = (A *)p->GetRuntimeClass()->CreateObject(); //创建一个与 p 所指向的对象同类的对象
```

12.2.3 应用框架

Visual C++围绕 MFC 提供了一些典型的 Windows 应用程序框架(framework)，这些框架提供了一些标准元素并自动建立起它们之间的联系，程序员的工作就是提供与具体应用相关的专门代码。

Visual C++通过提供 AppWizard,Resource Editor 以及 Class Wizard 来帮助程序员完成相关的工作。

1) AppWizard
2) Resource Editor
3) Class Wizard

12.3 习题