“C++ 程序设计”

笔　记

（ 课程名：计算引论　　）

（本笔记中的例题均在 VC++ 6.0 环境下通过）

1 C++ 入门

1.1 从 C 到 C＋＋

1980 年：贝尔实验室开始对 C 进行改进和扩充－－－带类的 C
1983 年：取名 C++
1994 年：制定 ANSI(美国国家标准协会:American National Standards Institute)C++标准草案
1994～：不断发展－－－目前的 C++

C＋＋：包括了整个 C：C 的全部特征、属性、优点
　　支持面向对象编程(OOP)
　　包括过程性语言部分和类部分
过程性语言部分：与 C 无本质上的区别
类部分：面向对象程序设计的主体

先是函数语言，面向对象语言

1.2 程序与语言

程序 ＝ 算法 ＋ 数据结构 ＋ 程序设计方法 ＋ 语言工具 ＋ 环境

C++语言的数据结构：以数据类型的形式体现。

程序：是软件
　　对机器而言：按硬件设计规范编制的动作序列(即：机器指令序列)
　　对人而言：　用语言(高、低级语言) 编写的语句序列
程序语言的发展：低级语言 → 高级语言

程序设计首要目标：可读性、易维护性、可移植性

1.3 结构化程序设计

是面向过程的程序设计
主要思想：功能分解、逐步求精
缺点：可重用性极差

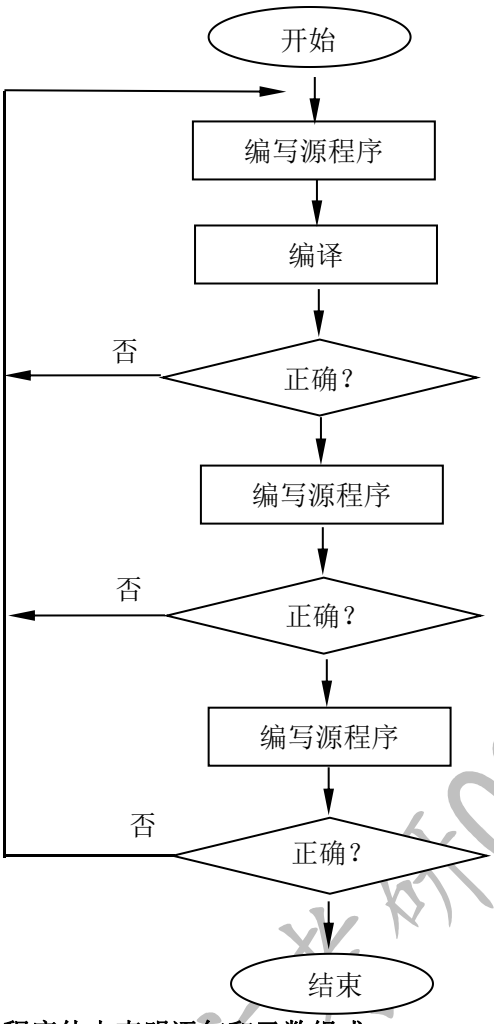
1.4 面向对象程序设计

基本思想：把信息和对这些信息的处理作为一个整体
C＋＋：是支持面向对象程序设计的语言
C＋＋的三大特性：封装性(数据隐藏)
　　继承性(软件重用)

多态性(函数重载)

1.5 程序开发过程

1.6 最简单的程序



程序体：程序体由声明语句和函数组成

C＋＋程序结构 { 注释：/* */ 和 //
编译预处理：#
程序体

例：//文件名：abc.cpp
```
/* 该程序的功能是：
   输出：I am a student. */
#include<iostream.h>
void main()
{ cout<<"I am a student.\n"; }
```

说明：

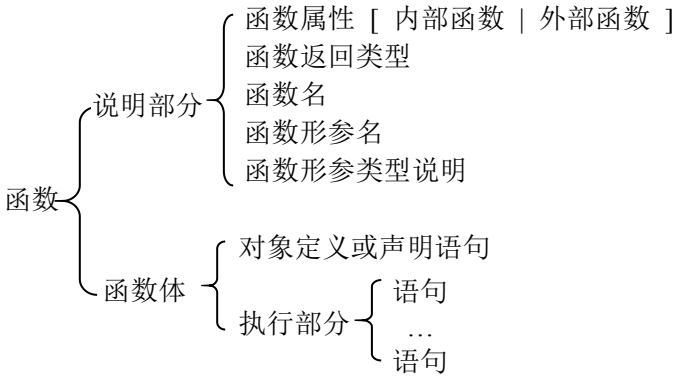
1. 大小写区分（即：abc 与 Abc 不同）
2. C＋＋系统是函数驱动的，一个可执行程序必须有、且只能有一个主函数：main()，作为程序的入口
3. C＋＋语句以分号（;）结束，一行可以写多条语句，一条语句可以分多行写
4. 所有的对象（变量）必须先定义，或先声明、后使用
5. 以双引号括起的为字符串常量，里边的"\"为转义字符

例：
```
#include <iostream.h>
int x=5,X=10;
```

```
void main()
{ int y=x*x;
  cout<<"y=x*x="<<y<<"  y=X*X="<<X*X<<
  "\n:::::::::::::::::: \n";
}              /* 执行结果：y=x*x=25  y=X*X=100
                  ::::::::::::::::::   */
```

1.7 函数

函数的组成：



函数格式：

函数属性标识符 函数返回类型标识符 函数名(形参说明列表)

{

•

•

•

}

其中形参说明列表：如果有多个形参，各形参之间用逗号（,）分隔

说明：

1. 一个函数如果没有返回值，则函数的返回类型标识符用“void ”代替，函数体中不必有 return 语句(如果有 return 语句,仅仅起提前返回的作用);否则用返回值的类型标识符,函数体中必须有 return 语句。
2. 函数与函数之间的关系是调用与被调用的关系，main()函数只能是调用函数，不可以被调用。
3. 一个函数内部不可以定义另一个函数，只可以调用另一个函数。
4. 函数只有定义或被声明后，才可以被调用。

函数声明：由函数原型加分号组成。例：int max(int,int);
函数原型：返回类型 函数名(形参类型,...,形参类型)

例：

```
#include <iostream.h>
int max(int,int);          //函数声明语句：函数原型加分号
void main()
{ int x=109,y=110;
  cout<<max(x,y)<<endl; //函数调用
}
int max(int x,int y)     //被调用函数定义
{ if(x<y) return y;
  else    return x;
}
```

函数：标准库函数
　　用户自定义函数

2 基本数据类型与输入输出

2.1 字符集与保留字

C＋＋中容许出现的字符：

26 个大写字母

26 个小写字母

10 个数字

其他符号：空格 + - * / = ， . _ : ；? \ ″ ’ ~ | ! # % & ( ) [ ] { } ^ <

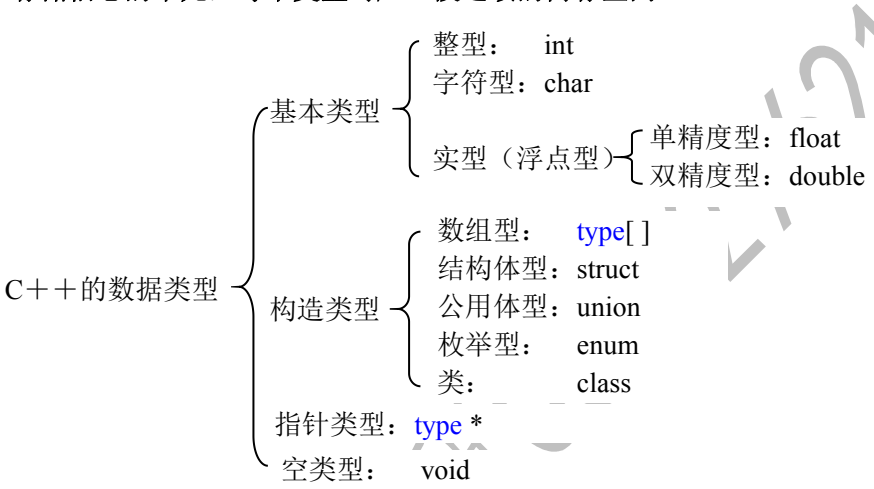
C＋＋保留字：

2.2 基本数据类型

数据(对象)：常量、变量

每个数据都具有类型，即：每个数据必属于某个类型。

变量：存储信息的单元，每个变量对应一段连续的内存空间



内存空间的最小单位：字节

数据类型的作用：编译器预算对象分配的内存空间大小

注意：数据无“void”类型，指针有“void”类型

不同的计算机，各类型的变量所占内存空间有所不同

bool 型：在 ANSI C++ 中具有

ANSI：美国国家标准协会（American National Standards Institute）

求某个类型的变量所占内存长度：利用运算符 sizeof

例：

```
#include <iostream.h>
void main()
{ short int x1;
  int x2;
  char y;
  //long double z;
  cout<<sizeof(x1)<<" "<<sizeof(x2)<<" "<<sizeof(y)<<" "<<sizeof(float)
  <<" "<<sizeof(double)<<" "<<sizeof(long double)<<endl;
}   //执行结果：2 4 1 4 8 8
```

2.3 变量定义

变量：在程序的运行期间，值会变、或值可变的数据量。

变量名：每个变量必须有名，即：变量标识符。

1. 命名变量名

规定：1 不能与 C＋＋的保留字、库函数名、类名、对象名同名
2 只能由 26 个大小写字母、数字和下划线组成
3 只能以字母或下划线开头

命名规则和程序书写风格：
变量定义
变量定义：每个变量必须被确定类型、变量名。
例如：int x,y;
注意：一条语句中只能定义同一类型的一个或多个变量

规定：变量必须“先定义、后使用”。程序编译时将分配存储单元，及进行语法检查。

唯一性：一个变量一个名，不同的名表示不同的变量。

2. 变量赋值与初始化

变量初始化：在定义变量的同时进行赋值
例：int x=5,y,z=9;
变量赋值：利用赋值运算符进行值的更新
例：int x=5; //x 初始化为 5
x=100; //把 100 赋给 x
x=x/2+7; //把 x/2+7 的计算结果赋给 x，即：x=57

3. 给类型起“别名”：typedef

```
例：#include <iostream.h>
    typedef short int I;  //把 I 作为 short int 的别名
    void main()
    { I x1;
      int x2;
      char y;
      cout<<sizeof(x1)<<" "<<sizeof(x2)<<" "<<sizeof(y)<<" "
          <<sizeof(float)<<" "<<sizeof(double)<<endl;
    }//执行结果：2 4 1 4 8
```

注意：typedef 并不产生新的类型

2.4 常量

常量：常数、或在程序运行过程中值始终不变的数据量。
例如：x = 5 + 19.7; 其中 5 和 19.7 就是常量
常量有类型：5---整型 19.7---实型 ‘a’---字符型

1. 整型常量

C＋＋提供 3 种表示整型常量的形式：
十进制： 123， －123， 0；
八进制： 以 0（零）开头的整数常量
例如：0123， 即：$0123_{8}=1*8^{2}+2*8^{1}+3*8^{0}=64+16+3=83_{10}$
-0123：表示八进制的负数 123，为十进制的-83。
十六进制：以 0x（零和 x）开头的整数常量
例如：0x123， 即：$0x123_{16}=1*16^{2}+2*16^{1}+3*16^{0}=256+32+3=291_{10}$
-0x123：表示八进制的负数 123，为十进制的-291。

```
例：#include <iostream.h>
    void main()
    { cout<<123<<" "<<0123<<" "<<-0123<<" "<<0x15<<" "<<-0x15
          <<" "<<0x123<<" "<<-0x123<<endl;
    }    //执行结果：123 83 -83 21 -21 291 -291
```

整型常量的类型：整型常量有类型，系统自动识别
在整型常量的后面加字母“L”或“l”（小写），表示该常量是长整型的。
注意：常量无 unsigned 型

2. 实型常量

C++ 提供 2 种实型常量的表示形式：
十进制：由数字和小数点组成。必须带小数点
例如：12.123、 0.0、 .0、 0.、 9453.13124
指数形式：数字 E(或 e)整数。E 前必须有数字，E 后必须是整数
例如：123E3（或：123e3） 即：123×10^3
1.234e5 1.234×10^5即：

有效位：单精度（float）－－－7 位
双精度(double) －－－15 位
实型常量在内存中所占字节数：以 double 型进行存储，占 8 字节。
但是，在后面加 f，则以 float 型存储，占 4 字节

```
例：#include <iostream.h>
    void main()
    { cout<<sizeof(1.23)<<"  "<<sizeof(1.23f)<<endl;
    }   //执行结果：8  4
```

3. 字符常量

用单引号括起的单个字符
特殊字符以“\”开头
例如：'1', '0', '/', '=', 'a', 'b', '?', '%'
'\n', '\t', '\\'
a='a';
a='A';
a='\101'; 8 进制的 101 等于 10 进制的 65，ASCⅡ值为 65 的字符是：A
所以，实际上：a='A';

注意：在内存中，字符以 ASCⅡ码存储，是以整数表示的
所以，字符和整数之间可以互相赋值

```
例：#include <iostream.h>
    void main()
    { int x1='A',x2='0',x3=0;
      char c=97;
      cout<<x1<<"  "<<x2<<"  "<<x3<<"  "<<c<<endl;
    }//执行结果：65  48  0  a
```

4. 字符串常量

用双引号括起的字符序列
例如："abc. |*88 住宅 ok"
"" ：空字符串
" " ：只有一个空字符的字符串
"a" ：只有一个字符 a 的字符串

字符串结束标志：'\0' （由系统自动添加）
注意：① "a"与'a'不同！(输出结果相同)
② C＋＋语言中无字符串变量！
③ 不能把字符串常量赋值给字符变量！例如：

```
char c;
c="a"; ×
```

④ 统计字符串长度时，不计字符串结束标志。例如：

"abcd"的长度为 4，而在内存中实际存放是：abcd\0

枚举常量

把变量的值一一列举，变量的值只能取其中之一

枚举类型的定义：

形式 1：enum 枚举类型名{元素 1，元素 2，…，元素 n}；
enum 枚举类型名 枚举变量名 1，枚举变量名 2，…；
例：enum city{上海,北京,南京,广州,天津}； //city 为枚举类型名
enum city city1,city2; //city1,city2 为 city 枚举类型的变量名

形式 2：enum 枚举类型名{元素 1，元素 2，…，元素 n}枚举变量名 1，枚举变量名 2，…；
例：enum city{上海,北京,南京,广州,天津}city1,cyty2,city3;

形式 3：可以直接定义枚举变量，而不定义枚举类型名
例：enum {上海,北京,南京,广州,天津}city1;

枚举类型的使用：

例：
```
#include"iostream.h"
enum city{Shanghai,Beijing,Nanjing,Tianjin=5,Guangzhou};
void ff(enum city x)
{ switch(x)
    { case 0:  cout<<"Shanghai\n";  break;
      case 1:  cout<<"Beijing\n";   break;
      case 2:  cout<<"Nanjing\n";   break;
      case 5:  cout<<"Tianjin\n";   break;
      case 6:  cout<<"Guangzhou\n"; break;
      default: cout<<"非法城市 !\n";
    }
}
void main()
{ enum city c1,c2,c3,c4;
  int i=7;
  c1=(enum city)i; //不能：c1=i;
  c2=Nanjing;
  c3=(enum city)5;
  c4=Shanghai;  /* 枚举变量的赋值：只能为列举元素之一 */
  ff(c1); ff(c2); ff(c3); ff(c4);
  cout<<c1<<" "<<c2<<" "<<c3<<" "<<c4<<endl;
}  /* 执行结果：非法城市 !
      Nanjing
      Tianjing
      Shanghai
      7 2 5 0   */
```

枚举变量的使用说明：

1. 先定义枚举类型，再定义枚举变量，然后使用变量
2. 枚举元素是常量，不是变量，故也称枚举常量；所以不能对枚举元素进行赋值
   上例中，不能：Shanghai=Beijing;
3. 枚举元素是常量，其常量值不是列举的“内容”，而是定义时的次序号：0,1,...,n
4. 枚举元素值在定义时可以“人为”指定，上例中：Tianjin=5，此后的元素均为 6,7,...
   上例中：枚举元素 Shanghai 的值为 0，Beijing 为 1，Nanjing 为 2，Tianjin 为 5，Guangzhou 为 6
5. 枚举变量的值只能取定义枚举类型时所列举的元素之一：
   例：c2=Nanjing;
6. 尽管枚举元素有值，但此值并不是整型值，所以不能把整型数赋值给枚举变量：
   不能：c1=5;

只能：c1=(enum city)5; /* 强制类型转换 */

7.枚举值（枚举元素）不是字符串

8.枚举值可进行逻辑判断比较运算：上例中，if(city1==Shanghai)
if(city2>Nanjing) :以序号值进行判断比较

2.5 常量定义 (const 常量)

符号常量：用一个标识符代表的常量，即：在程序中用 #define 命令定义某个符号所代表的常量。

例如：#define W “女”
#define M “男”
#define PRICE 123.789

注意：① 一旦定义了某个符号代表一个常量，则该符号在其作用域内就表示这个常量。
例如：x = PRICE * 0.8;
② 符号常量不是变量，所以，在其作用域内不能被赋值！
例如：PRICE = 123.012;
③ 为提高程序的可读性，符号常量一般用大写表示。

const 常量：冻结变量

例：const double pi=3.14159265;

注意：变量一旦被 const 限定，就变成不能改变值的常量了；因此，const 常量必须在定义时赋初值，并且 const 常量不能作左值！

符号常量和 const 常量的区别：

符号常量并不是变量，仅仅通过预编译命令进行“替换”，而 const 常量是占有内存的被“冻结”了的变量，C＋＋中使用 const 常量而不提倡用符号常量。

2.6 I/O 流控制

头文件：#include "iostream.h"

1. I/O 控制流的书写格式

I/O 控制流：输入或输出的字节序列（字符序列）

操作符：<< 向输出流 "cout"中插入字符
>> 向输入流 "cin" 中抽取字符

实际上，<< 、>> 是重载操作符
cout：C++ 预定义的标准输出设备
cin ：C++ 预定义的标准输入设备

作用：>> ：用从标准输入设备上所接收到的数据去更新操作符 ">>" 右边的对象
<< ：用操作符 "<<" 右边的内容输出到标准输出设备上

例：以默认的格式进行输入输出：

```
cout<<"How are you!";
cout<<a[0]<<"  "<<a[1]<<endl;  //或者：cout<<a[0]<<"  "<<a[1]<<'\n';
cin>>x;
cin>>a>>b>>c;
```

2. 使用控制符进行输入输出

控制符

使用控制符：

例：

```
#include<iostream.h>
#include<iomanip.h>  //格式控制符
void main()  //以指数形式输出：
{ cout<<setiosflags(ios::scientific)<<9999.999<<endl; }
```

3. 控制浮点数值的显示

控制符：setprecision(n:　　n 控制显示的数字位数（ 系统默认：有效位 6 位 ）
在用浮点表示的输出时：n 表示有效位数（ 系统默认：有效位 6 位 ）
在用定点表示的输出时：n 表示小数位数
在用指数形式输出时：　n 表示小数位数

4. 设置值的输出宽度

控制符：setw(n)
说明：仅仅对下一个数值的输出有效
　　如果实际位数>n，按实际宽度输出

5. 输出 8 进制和 16 进制数

dec：10 进制
oct：8 进制
hex：16 进制

6. 设置填充字符

setfill('填充字符')

7. 左右对齐输出

setiosflags(ios::left) 和 setiosflags(ios::right)

8. 强制显示小数点和符号

I/O 输出流对 2.0 以 2 显示，如果要输出小数，则用 setioflags(ios::showpoint)控制符
　　如果要输出正号，则用 setioflags(ios::showpos)控制符

2.7 函数 printf() 和 scanf()

11.5.1　　格式输出函数：printf()

形式：printf(“格式控制串”[，输出实参列表])
功能：按指定的格式输出数据
例如：printf(“上海交通大学：男生%d 人，女生%d 人”,n,w)
说明：输出实参列表中如果有多个参数，则用逗号分隔
　　实际上，printf()函数的格式为：
　　printf(实参 1，实参 2，...，实参 n)
格式说明：由字符‘%’和格式字符组成
格式字符：d、o、x、u、c、s、f、e、g 和附加格式说明符
说明：格式控制串中，除格式说明外，有什么输出什么
注意：①数据类型应与格式说明匹配，否则出错（但编译通过）
　　②格式说明个数应与参数列表中的个数相同，且次序对应：
　　.如果输出参数列表中的参数个数大于格式说明数，则参数列表中后面的参数值不被输出
　　.如果格式说明数大于输出参数的个数，则多出的格式说明会导致输出随机数
　　③格式说明串中可以用转义字符，例如：\n
　　④如果要输出字符'%'，则在格式串中的对应位置处写：%%
　　⑤不同的系统在执行格式输出时，可能会小有差别
　　⑥l 用于长整型输出，如果应该长整型数用%d 格式输出而不用%ld 格式，则编译通过，执行时输出的数出错，且在其后面的参数值均错
　　⑦缺省情况下，输出是右对齐、前补空格，但用‘-’附加格式说明符，将左对齐、后补空格
　　但是：字符串格式 s，则相反

11.5.2　　格式输入函数：scanf()

一般形式：scanf(格式控制串，地址列表)
功能：从标准输入设备接受输入的任何类型的多个数据

其中格式控制串与 printf()函数的格式控制串类似

格式符：P/31

注意：①变量的地址列表，而不是变量名

②数组变量的名就是该数组的地址，不必加'&'

③格式控制串中，除格式说明外的所有字符，在输入时必须对应地输入

④%lf：double 型输入

⑤用格式%c 时，空格、转义字符均有效；用格式%s 时，接受的字符以输入的非空字符开始至空字符或回车符止；以回车符开始执行接受输入的操作

⑥输入数值时不能规定精度

⑦输入数值时，遇非数值字符自动认为输入结束，遇宽度结束

⑧如果定义输入的宽度，则输入时必须小于等于宽度；否则，如果后面有其他输出数据，这些数据值将全部出错：

```
例：#include "stdio.h"
    void main()
    { int x1,x2,x3;
      char a[10];
      scanf("%2d,%3d,%d",&x1,&x2,&x3);
      printf("x1=%d  x2=%d  x3=%d\n",x1,x2,x3);
      scanf("%s",a);
      printf("a[10]=%s",a);
    }  /*  文件名：scanf 函数.exe
           如果输入：111,2,34567  则输出：x1=11  x2 和 x3 的值出错
                                  而 a[10]=1,2,34567  */
```

```
例：  #include"stdio.h"
      void main()
      {int a[10],i;
       printf("请输入 10 个整数：\n");
       for(i=0;i<10;i++)
          scanf("%d",&a[i]);
        for(i=0;i<10;i++)
          printf("\na[%d]=%d",i,a[i]);
      }
      /* 执行 1：输入：1 2 3 4 5 6 7 8 9 0   执行正确
         执行 2：输入：1,2,3,4,5,6,7,8,9,0   执行不正确 */
      说明：对于用循环进行输入，之间用空格分隔
```

```
例：#include"stdio.h"
    void main()
    {   double a;
        printf("请输入 1 个数：");
        scanf("%lf",&a);
    printf("\na=%f\n",a);
    }
    说明：对于 double 型，输入时应用%lf；如果用%f 则出错
          对于 double 型，输出时用%f 即可，系统会自动判断
```

3 表达式和语句

3.1 表达式

1. 表达式概述

表达式：操作符、操作对象组成的符号序列

2. 左值和右值

左值：左值表达式简称左值，能出现在赋值运算符左边的表达式
即，表示一个对象标识（有确定地址的对象）的表达式称为左值表达式
右值：只能出现在赋值表达式右边的表达式（例，常量只能作右值）
说明：左值既可以出现在赋值运算符的右边，又可以出现在左边。

3. 操作符的优先级和结合性

4. 语句与块

C++语言中，除了控制语句、函数调用语句外，几乎所有的操作都是通过表达式进行的。
语句以分号(;) 结束
C++ 语句分成 6 大类：

- 声明语句
- 控制语句
  - if () ~ else ~ （条件语句）
  - for() ~ （循环语句）
  - while() ~ （循环语句）
  - do ~ while() （循环语句）
  - continue （结束本次循环的执行）
  - break （终止 switch 或循环语句的执行）
  - switch （多分支选择语句）
  - goto （无条件转向语句）
  - return （从函数内返回语句）
- 函数调用语句：函数； 例：printf(“ok”);
- 表达式语句： 表达式；例：a=3; x+3; 5+5; 7;
- 空语句： ;
- 复合语句：用一对花括号把若干条语句括起来成为一条语句(语句块)
  说明：一对花括号后面的分号可以不写，
  但是复合语句内的最后一条语句的分号必须！

例：合法语句：
4;
a;
a+1-7;
a=1;

非法语句：如果 a=1
3=a+2;
5(7+a);
a+6=7;
a-a=0;

C＋＋程序书写规定：一行可写多条语句，一条语句可写多行（行尾加转移字符：'\'，表示连行）
注意：输入源程序时中、英文要及时切换，例如，中文的分号与英文的分号不是同一个字符
说明：C++ 程序中的注释部分不是语句，两种注释方法：
① 用 /* */ 括起的内容，不管是否换行，均是注释
② 用// 开头，至本行尾的内容是注释

3.2 算术运算和赋值

1. 基本运算符：+ - * / %

+：加法或正号
-：减法或负号
*：乘法
/：除法

%：模（整除取余，操作符两边的操作数必须为整型数）

说明：两个整数相除，结果为舍去小数部分、截取整数部分；

但是，如果有一个操作数是负数，则不同的机器可能有所不同，大多数采用“向零取整”法。

2. 复合运算符

+=　-=　*=　/=　%=

a=a+6　与：a+=6 等价

注意：a+=b+c　应：a=（b+c）

3. 算术表达式、运算符的优先级和结合性

1.算术表达式

由算术运算符、括号、操作数按语法规则相连接的式子

操作数即操作对象：常量、变量、函数等

例：　a*b/c-25+‘a’-(18*c+b)+max(a,b)

2.算术运算符的优先级和结合性

规定：优先级－－－先括号，再先乘除、模，后加减

结合性－－－从左到右

结合性即结合方向：当一个运算对象二侧的运算符级别相同时，规定的处理原则

例如：c-a+b

操作数 a 的两边的运算符相同，则先进行 c-a 操作

除 1 目、3 目、赋值运算符外的操作符的结合性均是从左到右

4. 溢出

即：数值范围

算术类型转换

5. 自动转换

1.字符型、整型、实型之间可以进行混合运算

其中字符型数据以 ASCⅡ码的十进制数参与运算

2.运算时，不同类型的数据要先转换成同一类型，然后进行运算

3.转换规则：P/38 的 图 3-1

类型转换由系统自动完成

例如：如果：　int i=9,ii;

long l=123;

float f=9.9;

double d=9.9;

则：l+’a’+ i*f - d/5 运算结果为 double 型

注意：①char、short 型数据参加运算，系统自动将其转换成 int 型，然后进行运算

②float 型数据参加运算，系统自动将其转换成 double 型，然后进行运算

③如果负的 int 型数据与无符号数进行运算，则负数作为无符号数进行运算：

```
#include"stdio.h"
void main()
{ int a=-1;
  unsigned b=0;
  printf("%d   %u\n",a+b,a+b);
} /* 输出：-1   4294967295  */
```

说明：-1 的补码为：1111111111111111

该码的无符号的十进制值为 4294967295

6. 强制转换

强制类型转换符：（类型）

将一个表达式的值转换成某个类型

格式：(类型名)(表达式)

说明：①括号必须，但是，如果表达式是单个操作数，其括号可以省

例如：(int) x

(int)(x+y)

②一般，系统对操作数自动进行类型转换，但如果：

a%b 要求 a 和 b 都是整型，如果不是，必须先进行转换：

(int)a%b (int)a%(int)b

③转换后只得到一个中间变量，被转换的变量的类型不变

3.3 增量和减量

自增自减运算符

1. 自增： ++

++i ：先加 1 后使用

i++ ：先使用后加 1

例：i=3;

j1=++i; j1=4 相当于执行：i=i+1; j1=i;

j2=i++; j2=3 相当于执行：j2=i; i=i+1;

2. 自减： --

--i ：先减 1 后使用

i-- ：先使用后减 1

例：i=3;

j1=--i; j1=2 相当于执行：i=i-1; j1=i;

j2=i--; j2=3 相当于执行：j2=i; i=i-1;

注意：①自增自减只能用于变量

②结合方向：自右至左

例如：a=-i++; 如果 i=6，则 a=-6，该语句执行完毕，i=7;

③常用于循环语句中的循环变量和指针变量中

有关表达式使用的说明

C++ 语言的表达式使用相当灵活，所以经常会出现一些问题

1.表达式中含有自增自减运算符

例：如果 i=3;

表达式 (i++)+(i++)+(i++) 的值为 9，该表达式运算毕,i 的值为 6

所以:a=(i++)+(i++)+(i++); b=i;

a 的值为 9，b 的值为 6

2.C 中有单目运算符和双目运算符，当难以确定时，编译时 C++ 尽可能多地按自左而右地组织运算符

例：i+++j; 编译时认为：(i++)+j;

3.在调用函数时，实参的求值顺序视机器系统而定，大多数的系统是右→左

例如：i=3;

cout<<i<<" "<<(i++);

左→右：输出：3，3

右→左：输出：4，3

3.4 关系与逻辑运算

1. 关系运算符和关系表达式

关系运算符和优先级

6 种关系运算符： < <= > >= == !=

同级（7 级） 同级（8 级）

关系表达式：用关系运算符把操作对象按语法要求联系起来的式子

被连接的对象可以是常量、变量、表达式。其中表达式可以是逻辑、赋值、关系表达式
关系表达式的值为逻辑值（真 / 假），可以参加运算

关系表达式的值 { 1———代表“真”
0———代表“假”

例：如果：a=1　b=2　c=3

① d=a>b==c>a+5

↓

d=(a>b)==(c>(a+5))

结果：关系表达式 (a>b)==(c>(a+5)) 的值为“真”，所以：d=1

② d=7/2+a>c+1==b>c-5

↓

d=((7/2+a)>(c+1))==(c>(a-5))

结果：关系表达式 ((7/2+a)>b)==(c>(a+5)) 的值为“假”，所以：d=0

2. 逻辑运算符和逻辑表达式

逻辑运算←→逻辑判断
3 种逻辑运算符：&&　　||　　!

逻辑表达式：用逻辑运算符把关系表达式或逻辑量连接起来的式子
逻辑量：常量、变量、及算术、逻辑、赋值表达式

逻辑判断 { 真———非 0：逻辑表达式的值为 1
假———0：　逻辑表达式的值为 0

a&&b：a 和 b 都为“真”时，表达式 a&&b 的值为“真”，否则为“假”
a||b：a 和 b 都为“假”时，表达式 a||b 的值为“假”，否则为“真“
!a：　a 为“真”时，表达式!a 的值为“假”，否则为“真”

例：① a=5>3&&2||8<4-!0

↓

a=((5>3)&&2)||(8<(4-(!0)))
1&&2　||　8<(4-1)
1　||　0

表达式 a=5>3&&2||8<4-!0 逻辑判断结果为“真”，所以该表达式的值为 1，所以 a=1。

② 如果：　a=1　b=96　c=3
d=a=’a’>b+!’0’<1&&c==3||c>a&&!b==b++

↓

d=a=(((’a’>(b+(!’0’)))<1)&&(c==3))||((c>a)&&((!b)==(b++)))
(97>96　<1)&&　1　||　1　&&　(0　==　96)
0　&&　1　||　1　&&　0

该逻辑表达式的判断结果为“假”，其值为：0，所以：d=a=0。

3.5 if 语句

逻辑判断语句

1. 3 种形式

① if(表达式) 语句
② if(表达式) 语句 1 else 语句 2
③ if(表达式 1) 语句 1
　else if(表达式 2) 语句 2

```
                    if(表达式) 语句
                    else if 语句
        .
        .
        .
    else if(表达式 n) 语句 n
    else 语句 n+1
```

注意：①表达式左右括号必须要
②凡是“语句”，后面必须有分号“；
③第 3 种形式可以理解成是第 2 种形式的嵌套

2. if 语句的嵌套

即：上述 3 种形式中的“语句”之一是 if 语句，就是 if 语句嵌套

例 1：某商场优惠顾客：购买金额大于等于 500 元的 8 折，大于等于 300 元及 500 元以下的 9 折，300 元以下的全部 9.5 折；编写计算实际应付金额的程序：

程序：

```
#include"stdio.h"
void main()
{ float x;
  printf("请输入总金额：");
  scanf("%f",&x);          /* 如果输入 500 */
  if(x>=500) x=x*0.8;      /* 符合条件，执行结果：x=400 */
  if((x>=300)&&(x<500)) x=x*0.9; /* 此时 x=400,符合条件，执行结果：x=360 */
  if(x<300) x=x*0.95;
  printf("\n 应付金额： %f\n",x); /* 输出：360 */
  getchar(); getchar();
} /*  文件名：if 语句_例 1_错误
      注意：此程序有毛病：①应付金额变量名不应与 x 同名！
                          ②输入 300，则输出：256.5
                          原因：变量 x 的值被改变和比较时的精度误差  */
```

改进后的程序：

```
#include"stdio.h"
void main()
{ float x,y;
  printf("请输入总金额：");
  scanf("%f",&x);
  if(x>=500) y=x*0.8;
  if((x>=300)&&(x<500)) y=x*0.9;
  if(x<300) y=x*0.95;
  printf("\n 应付金额： %f\n",y);
  getchar(); getchar();
} /*  缺点：一旦执行了，后面的 if 语句就不应再判断了，再次改进： */
```

再次改进后的程序：

```
#include"stdio.h"
void main()
{ float x,y;
  printf("请输入总金额：");
  scanf("%f",&x);
  if(x>=500.0) y=x*0.8;   //对于两个浮点数，无法精确判断是否相等
  else if((x>=300.0)&&(x<500.0)) y=x*0.9;
  else y=x*0.95;
  printf("\n 应付金额： %f\n",y);
```

```
    getchar(); getchar();
}  /*  文件名：if 语句_例 1    */
```

例 1. 对任意输入的 3 个数，按大小顺序输出：

```
程序：#include"stdio.h"
      void main()
      { float x1,x2,x3,x;
        printf("请输入 3 个数，之间用逗号分隔：");
        scanf("%f,%f,%f",&x1,&x2,&x3);
        if(x1<x2)
          { x=x1; x1=x2; x2=x; } /* 从 x1 和 x2 中确定大的数→x1   */
        if(x1<x3)
          { x=x1; x1=x3; x3=x; } /* 比较 x1 是否比 x3 大？  确定大的数→x1   */
        if(x2<x3)
        { x=x2; x2=x3; x3=x; }   /* 从 x2 和 x3 中确定大的数→x2   */
        printf("%f %f %f\n",x1,x2,x3);
      }
```

3.6 条件运算符

格式：表达式 1 ？表达式 2 ：表达式 3

运算规则：如果表达式 1 为“真”，整个表达式的值取表达式 2 的值，否则取表达式 3 的值

例：① x=3>4?4:5/2

↓

x=((3>4)?4:(5/2))

x=2

② 如果 a=4:

a=a*5,a*4,a==4?a/2:++a

↓

a=a*5,a*4,(a==4?a/2:++a)

执行结果：表达式 a=a*5,a*4,a==4?a/2:++a 的值为：21，a 的值也是 21

如果：表达式是 a=a*5,a*4,a=4?a/2:++a ，则执行结果为 10，a 的值是 10

③ 如果 a=4:

b=(a*5,a*4,a=4?a/2:++a)

执行结果：表达式 b=(a*5,a*4,a=4?a/2:++a) 的值为：2，a 的值是 4

b 的值是：2

3.7 逗号表达式

1. 逗号运算符

，：也称：顺序求值运算符

作用：连接表达式

2. 逗号表达式

格式：表达式 1，表达式 2，…，表达式 n

求解过程：先求表达式 1，再求表达式 2，…，最后求表达式 n

整个表达式的值为表达式 n 的值

例：如果 a=5：表达式 a=3*5,a*4 的值为 60

即：a=3*5,a*4

注意：表达式 a=3*5,a*4 与 a=(3*5,a*4) 的区别

例：如果 a=5：表达式 3*5,a*4 的值为 20

逗号表达式可以嵌套：

例：如果 a=5：表达式 (a=3*5,a*4),a+5 的值为 20

即：a=3*5,a*4 这时的 a=15

所以：a+5 就等于 20

3.8 求值次序与副作用

不同的编译器求值顺序不同

p/49

消除副作用：将一个表达式分解成多个表达式，或加括号

4 过程化语句

即流程控制语句

4.1 while 语句（“当”型循环结构）

一般形式：while(表达式)　语句

说明：1. 先判再执行

2. 表达式必须有；表达式可以是常量或常量表达式，但出现“死循环”

例：...

```
while(5)
{...
}
...
```

例：求 1 加到 100 的和

```
#include “stdio.h”
void main()
{ int i=1,sum=0;
  while(i<101)
    { sum=sum+i;  ++i;  }
  cout<<"sum= "<<sum<<endl;
}
```

4.2 do ~ while 语句

一般形式：do 语句

while(表达式);

说明：先执行、后判断

注意：该语句后面的分号必须要有！

例：求 1 加到 100 的和

```
#include “stdio.h”
void main()
{ int i=1,sum=0;
  do { sum=sum+i;  ++i;  } while(i<101);
  cout<<"sum= "<<sum<<endl;
}
```

4.3 for 语句

一般形式：for(表达式 1；表达式 2；表达式 3) 语句

⇩ 可理解为：

for(循环变量赋初值；循环条件；循环变量增值) 语句

例：求 1 加到 100 的和

```
#include “stdio.h”
void main()
{  for(int i=1,sum=0;i<101;++i)
       sum=sum+i;
  cout<<"sum= "<<sum<<endl;
}
```

说明：1. for 语句中的 3 个表达式之间用“;”分隔
2. 执行过程：
3. 3 个表达式都可以省，但是分号不能省
表达式 1 省：循环变量的初值在 for 语句前已赋过
表达式 2 省：不判断循环条件－－－出现“死循环”
表达式 3 省：循环变量的“增值”操作在循环体中进行，否则，如果终止条件不出现，则出现“死循环”
4. 表达式 1 在 for 中只执行一次
5. 在表达式 1 中可以对循环变量赋初值，也可以利用逗号表达式对无关的变量赋初值
例：for(sum=0,j=0,i=1;i<101;++i)
6. 表达式 2－－判断用，所以常常是关系、逻辑表达式，但也可以出现其他表达式
例：for(x=1,i=0;y=x*x,x<=5;++i,++x)

4.4 switch 语句（多分支选择语句）

一般格式：switch(表达式)
{ case 常量表达式 1：语句段 1
case 常量表达式 2：语句段 2
.
.
.
case 常量表达式 n：语句段 n
default: 语句
}

例：
```
#include "stdio.h"
void main()
{ int x;
  cout<<"请输入一个整数(可以是负数)：";
  cin>>x;
  switch(x)
  { case 1:   cout<<"1：\n";
              cout<<"   上海交通大学 \n";
    case 2:   cout<<"2\n";
    case 3:   cout<<"3\n";
    case 4:   cout<<"4\n";
    case '5': cout<<"字符 5\n";  /* 字符'5'的 ASCⅡ码 ：53  */
    default:  cout<<"default \n";
  }
} /*  执行：分别输入 1、3、5、53、-1，  讲解：应使用 break 语句  */
```

改进后的程序：

```
#include "stdio.h"
void main()
{ int x;
  printf("请输入一个整数(可以是负数)：");
  scanf("%d",&x);
  switch(x)
  { case 5-4: cout<<"1：\n";
              cout<<"   上海交通大学 \n";  break;
    case 2:   cout<<"2：\n";  break;
    case 3:   cout<<"3：\n";  break;
    case 4:   cout<<"4：\n";  break;
    case '5': cout<<"字符 5\n";    break;  /* 字符'5'的 ASC 码 ：53  */
    default:  cout<<"default \n";  break;
  }
} //  执行：分别输入 1、3、5、53、-1
```

说明：①每个语句段可以由多条语句组成，而不必加花括号
②switch(表达式)，括号不可以省；表达式可以是：常量，变量，关系、逻辑、算术、赋值表达式，但是表达式的值必须是整型或字符型
③default 可以不写：都不符合，就什么也不做
④switch 语句只判断一次，作为寻找执行该语句的入口点
⑤case 常量表达式： 的作用：仅仅作为语句标号
⑥利用 break 语句可以跳出 switch 语句
⑦多个 case 可以共用一组执行程序段，例：

```
…
case 1:
case 7:
case 9:          a=a+b;  cout<<"a="<<a;  break;
case 'a'+'p':
case 100:        max(a,b);  break;
…
```

即：符合 1、7、9 的情况，均执行同一段语句；符合'a'+'p'和 100，均执行同一段语句

4.5 转向语句

1. break 语句

作用：1.中止 switch 语句的执行，并跳出 switch 语句
2.从循环体中跳出，转而执行循环体的下一条语句
注意：只能用于 switch 和循环语句中！

2. countinue 语句

作用：结束本次循环
注意：并不从循环体中跳出，除非已处于循环结束点
例：…

```
for(j=0;j<100;++j)
 { a=a+j*j;
   if(j==50)  coutinue; /* 如果 j 等于 50 就不执行下面的语句,直接进行下一轮循环 */
   cout<<"a="<<a;
 }
```

3. goto 语句

一般形式：goto 语句标号；
说明：①goto 为无条件转向语句，只有与 if 语句一起使用方可构成循环
例：求 1 加到 100 的和

```
#include "stdio.h"
void main()
{ int i=1,sum=0;
  loop:sum=sum+i;
       ++i;
       if(i<101)  goto  loop;
  cout<<"sum= "<<sum;
}
```

③ 应该禁用 goto 语句（不符合结构化原则）

4.6 实例

例 1： p/66 用公式： $\frac{\Pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots\cdots$

求Π的近似值，直到最后一项的绝对值小于： 10^{-6}

分析：

$$\frac{\Pi}{4} = \frac{1}{a_0} - \frac{1}{a_1} + \frac{1}{a_2} - \frac{1}{a_3} + \frac{1}{a_4} - \cdots\cdots \frac{1}{a_i}$$

$$a_i = s \times (a_{i-1} + 2)$$

规律：①可以看作：

当 i=偶数时，s 取 1；i=奇数时，s 取 -1 （ s=1 或 s=-1 ）

②特殊情况：

$a_0 = 1$

程　　序　　：

```
#include"stdio.h"
void main()
{ int s;
  float a,pi;
  for(s=1,a=1,pi=0;;a=a+2)
  { pi=pi+s/a;
    s=-s;
    if(1/a<1e-6)  break;
  }
  cout<<"pi="<<pi*4<<endl;
}
```

请改用 do-while 和 while 语句编写该程序

例 2：“母牛”问题

```
/* 有一头小母牛，自第 4 年起每年生一头母牛，问：到 n 年共有多少头母牛？  */
//1: 用数组实现
#include<iostream.h>
#define N 30
void main()
{   char c;
    int i,n,num[N]={0,1,1,1}; //初始化：第 0 年 0 头牛、第 1 年 1 头牛、…、第 3 年 1 头牛
    do{ cout<<"\n\n 请输入年数 (大于 3、小于 30) : ";
        do{ cin>>n;
if(n<4 || n>29) cout<<"必须输入大于 3、小于 30 的数，请重新输入：";
          }while(n<4 || n>29);
        for(i=4;i<=n;i++)    //第 4 年开始计算、至第 n 年
            num[i]=num[i-1]+num[i-3]; //第 i 年是 i 的前一年的牛与 i 的前 3 年的牛数之和
             //即：i 的前 3 年有多少牛、到 i 年就生出多少头牛，加上前一年的牛数就是 i 年的牛数
        cout<<"\n 第"<<n<<"年共有 "<<num[n]<<" 头母牛\n";
        cout<<"\n 继续 ? (Y/N):  "; cin>>c;
      }while(c=='y'|| c=='Y');
}

//2: 用递归实现
#include<iostream.h>
int produce(int n)
{ if(n<4)  return 1;  else  return(produce(n-1)+produce(n-3)); }
void main()
{   int n; char c;
    do{ cout<<"\n\n 请输入年数 (大于 3、小于 30) : ";
        do{ cin>>n;
            if(n<4 || n>29) cout<<"必须输入大于 3、小于 30 的数，请重新输入：";
          }while(n<4 || n>29);
        cout<<"\n 第"<<n<<"年共有 "<<produce(n)<<" 头母牛\n";
        cout<<"\n 继续 ? (Y/N):  "; cin>>c;
      }while(c=='y'|| c=='Y');
}
```

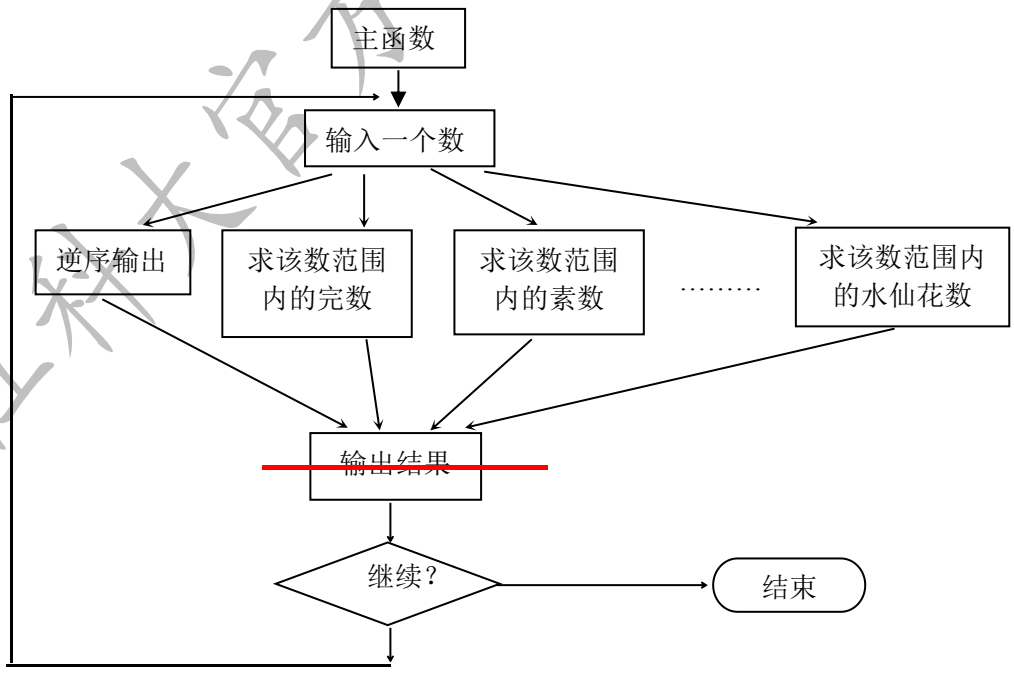
5 函数

5.1 函数概述

程序－－－功能模块
程序－－－子程序
C++ 语言： 程序－－－函数

程序结构：

例： 编写一个对某个正整数有关的处理程序



注意：层次结构、函数调用关系、函数返回值

说明：1.源程序文件：一个源文件是一个编译单位，不是以函数为单位进行编译的
2.C++ 程序由一个或多个源文件组成；分多个文件可以提高编译和调试的效率

3.主程序只能有一个，程序执行自主程序开始，由主程序结束而结束
4.函数必须定义，定义时各函数相互独立：不能嵌套定义；除主函数外，函数间可以相互调用
5.函数分类：从用户角度：库函数、自定义函数
从形式：无参函数、有参函数

5.2 函数原型

函数声明就是函数原型
函数的声明和函数的定义，在返回类型、函数名、参数个数、参数类型、及参数顺序上必须完全一致，否则编译时出错
C： 函数抽象用于表达问题求解中的一个过程
C++：函数抽象主要用于定义类上的操作接口

5.3 全局变量与局部变量

1. 程序的内存区域

代码区、全局区、堆区、栈区

2. 全局变量

在函数外部定义的变量称全局变量
作用域：从被定义的位置起，至文件结束处止
说明：全局变量在程序执行的整个过程中，占用固定的内存单元
具有初始值
可以被在其定义后的所有函数共同使用（解决“一个函数只能返回一个值”的问题）
在同一源文件中，如果外部变量与局部变量同名，则：在局部变量的作用域内，同名的全局变量不起作用

3. 局部变量

只在某一函数内部定义的变量称局部变量，局部变量存放在栈区
作用域：只局限于本函数内
说明：不同函数的局部变量可以同名
形参是局部变量
除本函数外，其他函数一概不能使用
复合语句中的局部变量：
复合语句可以看成是“程序块”，在其内部可另行定义变量－－－分程序局部变量
这种变量出了该复合语句就无效
一般：在复合语句中的循环变量可以在复合语句中定义

例：
```
#include"stdio.h"
void main()
{ int i=0,j=9;
  { int j=0;          //j 的作用域：自此起，到本复合语句结束处止
    printf("复合语句 j=%d\n",j);
  }
  printf(" j=%d\n",j);
  for(int k=0;k<5;++k) //k 的作用域：自此起，到本函数结束止
     printf(" k=%d  ",k);
  printf("\nk=%d\n",k);
} /* 执行输出结果：复合语句 j=0
                  j=9
                  k=0  k=1  k=2  k=3  k=4
                  k=5                    */
```

5.4 函数调用机制

栈 函数的整个过程就是栈空间操作的过程：
保护调用函数的运行：入栈

建立被调用函数的栈空间
传递参数
将控制转给被调用函数
……

静态局部变量

静态局部变量：局部变量定义时，前面冠以 static
注意：静态局部变量只在函数第一次被调用时进行初始化
可见性：本函数内

存储类型：auto ———自动（动态，局部变量）
static———静态（可以全局，也可局部变量）
register——寄存器（形参，局部变量）
} 用作定义
extern ——外部的全局变量 —— 仅作说明

C++规定：①每个变量和函数有两个属性：存储类型和数据类型，并必须加以定义：
存储类型 数据类型 变量名列表；
②局部变量的存储类型缺省，默认为：auto 型
③外部变量：指本文件外的全局变量
如果外部变量的定义出现在本源文件、且在引用前，则引用时可以不说明，否则必须说明
外部变量的定义：只定义一次，且在函数外定义

5.5 递归函数

递归调用：直接或间接地调用自己
直接递归调用：在函数 f 中调用函数 f

例如：...

```
void f()
{ ...
  f();
  ...
}
```

间接递归调用：函数 f 调用函数 ff，而函数 ff 调用函数 f

注意：要有终止递归调用的条件！

例 “汉诺”问题
汉诺问题：a 针上有从小到大排列的 n 个盘，要求把这些盘全部搬到 c 盘上，可以利用空的 B 针；
但是：每次只能搬动一个盘，且在搬动时每个针上必须保持盘从小到大的排列次序。
分析：1 如果 n = 1，则盘：a --> c；执行步骤 3；
2 否则 n > 1，则把盘分成最大的一个和其他的（n-1）个，并把（n-1）当作一个盘：
① (n-1)个盘：借助 c 针，由 a 搬到 b,
② 把最大的一个盘：由 a 搬到 c，
③ 再把(n-1)个盘：借助 a，由 b 搬到 c，
3 结束。

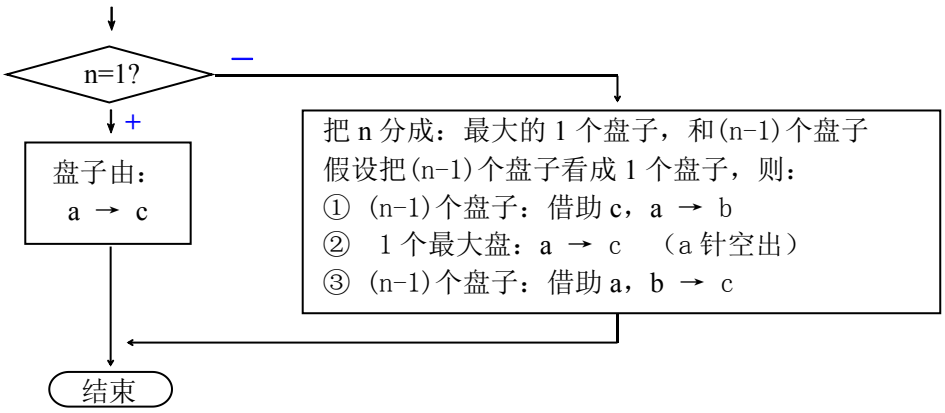
设子函数：
void move(源针，目标针)：把一个盘由源针搬到目标针

void hanoi(盘数，源针，借助针，目标针)：
把参数“盘数”指定的这些盘，通过借助针，由源针搬到目标针

分析：
↓
输入盘子数:n
↓
调用 hanoi()函数
↓

结束

hanoi()函数：(借助 b 针，把 a 针上的所有盘子 → c 针上)



其中：①可以把(n-1)个盘子再当作 n 个盘子：再分成最大的 1 个盘子和(n-1)个
进行上述的移动，直至 n=1 个 （递归）
②完成“移动”操作的函数：void move(源针，目标针)
“汉诺”函数：hanoi(盘子数，源针，借助针，目标针)
③ (n-1)个盘子：借助 a，b → c

程序：

```
#include"iostream.h"
void move(char a,char c)
{ static int i;  i++;
  cout<<i<<" :  "<<a<<" → "<<c<<endl; }

void hanoi(int n,char a,char b,char c)
{ if(n==1)  move(a,c);
  else
    { hanoi(n-1,a,c,b);
      move(a,c);
      hanoi(n-1,b,a,c);
    }
}

void main()
{ int num;
  cout<<"请输入盘子数：";
  cin>>num;
  cout<<"按\"汉诺塔\"的规则，把"<<num<<"个盘子从 a 针搬到 c 针的步骤是：\n";
  hanoi(num,'a','b','c');
} /* 如果输入 4：(如果输入 5，则执行 31 步)
     按"汉诺塔"的规则，把 4 个盘子从 a 针搬到 c 针的步骤是：
     1:  a → b
     2:  a → c
     3:  b → c
     4:  a → b
     5:  c → a
     6:  c → b
     7:  a → b
     8:  a → c
     9:  b → c
```

```
10: b → a
11: c → a
12: b → c
13: a → b
14: a → c
15: b → c  */
```

5.6 内联函数

inline：也称内置函数、内嵌函数
目的：减少开销、提高效率
在编译时，象对待“宏”一样进行替换，程序运行时没有函数调用过程和函数返回
定义、声明时：在前面冠以“inline”即可
说明：
⑴ 如果被调用函数定义在被调用之前，且定义时冠以 inline，则以后每次被调用均作内联函数处理。

如果被调用函数定义在被调用之后：
①定义时没有冠以 inline：如果在被调用前声明过该函数，且声明时冠以 inline，则以后每次被调用均作内联函数处理
②定义时冠以 inline：如果在被调用前声明过该函数，但声明时没有冠以 inline，则被调用均不作内联函数处理
即：必须声明成“内联”的，或无声明、但定义成“内联”的

⑵内联函数内不能出现 switch、循环语句，否则报错，作普通函数进行调用。

例：
```
#include<iostream.h>
inline void abc()
{ for(int i=1;i<10;++i) cout<<i<<endl; }
void main()
{ cout<<"start...."<<endl; abc(); cout<<"end..."<<endl; }
```
该程序能够通过编译，但必须注意在内联函数中不应该出现循环语句。

⑶递归函数不能被定义成“内联”函数
⑷内联函数不能进行“异常接口声明”，例如，被除数为 0 的“异常声明”
⑸对常被调用的小而简单的函数进行“内联”。

5.7 重载函数

1. "重载"概念

面向对象系统的 3 大特性之一的多态性就是通过重载实现的
重载：即同名不同义，在不同的情况下可表现出不同的“行为”
重载包括：
函数重载：多个函数同名，但执行的操作和操作对象不同
操作符重载：一个运算符，可进行多种不同含义的解释，进行不同的操作
问题的提出：
例如，比较大小：2 个整数比较、2 个浮点数比较、.....
在 C++ 中必须逐个定义不同名的多个函数
希望：由系统自动判断，用户只要记住函数名即可
重载：写多个同名函数，由系统根据操作数自动判断究竟调用哪个函数

2. 匹配重载函数的顺序

原则：
按如下先后顺序：
1 寻找一个完全匹配，找到就调用
2 通过内部转换、然后寻找严格匹配，只要找到就调用
3 通过用户自己定义的转换寻找严格匹配，找到就调用

1. 重载函数至少在参数个数，参数类型、或参数顺序上有所不同，仅返回类型不同不行
2. 不能用 typedef 定义的类型名来区别重载函数中的参数类型，因为“编译”时不区分其差别
3. 同名函数应具有相同的功能

例：// 5.8 节 一般函数名重载

```
#include<iostream.h>
#include<stdio.h>
int add(int x,int y)
     { return x+y; }
double add(double x,double y)
     { return x+y; }
int add(int x,double y)
     { return int(x+y); }
double add(double x,int y)
     { return x+y; }

void main()
{ cout<<add(99,88)<<"  ";
  cout<<add(99.9,88.8)<<"  ";
  cout<<add(99,88.8)<<"  ";
  cout<<add(99.9,88)<<endl;
}  /* 执行结果：187  188.7  187  187.9   */
```

说明：本例如果只定义一个函数：double add(double x,double y)；本程序被正常执行。
但是一旦出现函数重载（定义 1 个以上的同名函数），就必须根据数据的情况逐一定义各个同名函数；例如，上例中有函数重载，但是，如果不定义函数：double add(double x,int y)；在编译时语句：cout<<add(99.9,88)<<endl;出错！

5.9 默认参数的函数

C++ 规定：实参与形参的个数必须相同，且类型相对应
但是，允许调用时不传递或部分不传递实参值，而采用缺省值

例：

```
#include "iostream.h"
void set(int a=0,int b=0,char c='?')
{ cout<<a<<"  "<<b<<"  "<<c<<endl; }
void main()
{ set(); set(999); set(123,456); set(111,222,'*'); }
/* 执行结果：0  0  ?
             999 0 ?
             123  456  ?
             111  222  *      */
```

说明：①当程序中的某个函数有缺省参数的函数声明，则在该函数的定义中不允许出现默认值
②一旦某个参数定义了缺省值，其右边的所有参数必须定义缺省值
③调用时，实参从左至右、逐一传递给形参
④当有函数重载、且重载的函数又有默认参数时，要注意函数调用的二义性问题：p/96(页底)
⑤默认值的限定：缺省值可以是常量、全局变量、函数调用的返回值，但不可以是局部变量：p/97

6 程序结构

多文件结构

6.1 外部存储类型

extern 声明：仅声明是本文件外的变量或函数，不是定义外部变量或函数

外部变量的定义：只定义一次，且在函数外定义；定义格式：与全局变量的定义格式同
外部变量的说明：可以说明多次
说明格式：extern 外部变量类型 变量名列表；
例：extern int a,b;
extern char x,y;

6.2 静态存储类型

1. 静态全局变量

全局变量定义时，前面冠以 static
作用：全局变量的作用域是整个程序，对于多文件程序结构，一个文件的全局变量可以被另一个文件使用，但静态全局变量的作用域被限定在本文件内，别的文件不能使用

2. 静态函数

函数定义时，前面冠以 static
作用：函数的作用域是整个程序，对于多文件程序结构，一个函数可以被另一个文件使用，但静态函数的作用域被限定在本文件内，别的文件不能使用。

6.3 作用域

作用域：标识符在程序中的有效范围

1. 局部作用域

局部作用域：自定义的位置起，至所在的语句块尾或函数尾止
例：

```
#include<iostream.h>
    void main()
    { for(int i=0,x=20,y=30;y<200;i++) //i 自此至本函数尾有效
        { int z=i+1;                   //z 自此至本语句块尾有效
          cout<<(y+=x)<<endl;
        }
      cout<<i<<endl;  //合法
      cout<<z<<endl;  //error：z 没有定义
    }
```

2. 函数作用域

一个标识符不管在函数的什么地方定义，自函数的起始点起，至函数的终止点都有效，则此标识符具有函数作用域。
语句标号是唯一具有函数作用域的标识符
p/108 例题：可以使用出现在后面的语句标号标识符

3. 函数原型作用域

指：函数声明 (不是定义) 中的形参标识符的作用域仅仅在函数原型中

例：

```
int max(int x,int y);
    x=100;  //error：x 没有定义
```

4. 文件作用域

文件作用域：本文件内有效，也称全局作用域
全局变量（包括静态全局变量）、静态函数具有文件作用域

6.4 可见性

可见即可用，可见性与作用域相一致
注意：内部标识符“支配”外部同名标识符
局部标识符“支配”全局同名标识符（解决办法：加域限定符 ::进行引用）

6.5 生命期

1. 静态生命期

与程序运行期同
函数、全局变量、静态局部变量都具有静态生命期

2. 局部生命期

非静态的局部变量具有局部生命期
如果不进行初始化，其值不确定

3. 动态生命期

动态分配的对象具有动态生命期：如果不释放（即收回），则直到程序运行结束，生命期才结束

6.6 头文件

以 .h 为扩展名的文件，用 #include 命令包含进来

6.7 多文件结构

一个大程序可以由多个源文件组成，每个源文件可以进行独立编译

6.8 预编译处理

编译预处理：C++ 的主要功能
在程序中通过预处理命令实现，不是 C++ 的语句
C++ 编译系统：先对程序中的特殊命令进行预处理，然后把预处理结果和源程序一起进行常规编译处理
（词法、语法分析，代码生成，优化等）

提供 3 种预处理功能：宏定义
文件包含
条件编译
} 均以“#”开头

1. #include 命令

2. 作用：把指定的源文件（或：头文件）的全部内容包含到当前文件中

一般形式：#include<文件名>
#include”文件名”
形式 1：直接按系统标准方式检索文件目录
形式 2：在当前源文件的目录中寻找，找不到再按系统标准方式检索其它文件目录

说明：①一条命令只能包含进一个文件
②被包含进来的文件的次序，就是该条包含命令的次序
③如果嵌套包含文件，则涉及到的全部文件均应包含进来，且最深一层被包含的文件所对应的包含命令应在最前面
例如：文件 3 中包含文件 2，文件 2 中又包含文件 1，应该：
#include “文件 1”
#include “文件 2”

#include “文件3”

④被包含的文件的扩展名： .h .c .cpp

⑤被包含后的所有文件均成为一个文件，并作为一个文件进行编译、连接，得到一个目标文件、一个可执行文件；但是：●作为源文件，依旧相互独立存在

●原各文件中的全局静态变量就成为整个文件的全局静态变量了（不必冠以 extern）

3. #define 命令

注意：在 C++ 中已经被 const 定义所代替

#define、#undef 命令

宏定义：宏代换

作用域：自#define 起，#undef 或文件结束止

注意：一次命令定义一个宏；是命令，不是语句；但可以出现在函数中

不带参数的宏定义：

作用：用指定的宏名（符号常量）代替一个字符串

形式：#define 宏名 字符串

例：#define Name 上海交通大学
#define Name1 “上海交通大学” } 注意：两者不同！

例：...

```
#define Name 上海交通大学
#define Name1 “上海交通大学”
void main()
{ char n[20]=Name;
  char m[20]=Name1;
  ... }
```

预处理后为：

```
...
void main()
{ char n[20]= 上海交通大学;      ——— 非法语句
  char m[20]= “上海交通大学”;    ——— 合法语句
  ...
}
```

说明：①一般用大写标识宏名

②宏名代表一个字符序列，预处理时不检查其合法性，在编译被宏展开后的源程序时进行合法检查

③宏定义不是语句，行末不必加分号，如果加了，则一起转换

④在宏定义中，可以利用已定义的宏

⑤用双引号括起的为字符串常量，在双引号中出现的宏名一概不替换

带参数的宏定义：

作用：除进行字符序列替换外，再进行参数替换

一般形式：#define 宏名(参数列表) 字符序列

宏替换过程：用指定的字符序列替换宏名，然后用实参替换参数列表中的形参

例：#define S(a,b) a*b+a/b

```
...
x=S(5,7);
...
```

⇒ 第一步：x=a*b+a/b;
第二步：x=5*7+5/7;

例：#define Pi 3.1415926
#define S(r) Pi*r*r

```
...
x=S(a+b)*x;
...
```

↓

```
#define Pi 3.1415926
#define S(r) Pi*(r)*(r)
...
x=S(a+b)*x;
...
```

↓

x=3.1415926*(a+b)*(a+b)*x;

```
x=3.1415926*a+b*a+b*x;
```

说明：①在宏名与参数列表的括号之间不能有空格，否则被理解为不带参数的宏替换
②“实参”与“形参”应一一对应，无类型而言，仅替换；“实参” 如果是表达式，并不进行运算
③与函数的形参和实参有本质上的区别：不分配内存、参数无类型、表达式不求值、宏展开在预编译时进行（函数调用在运行时进行）
④宏替换不占运行时间，只占编译时间

4. 条件编译命令

作用：对满足条件的部分程序内容进行编译，否则不参与编译
功能：便于调试

判断符号常量，进行条件编译：
形式 1：#ifdef 标识符
程序段
endif
作用：如果标识符被定义过（一般用#define 命令定义），则编译程序段，否则不编译
形式 2：#ifdef 标识符
程序段 1
#else
程序段 2
endif
作用：如果标识符被定义过，则编译程序段 1，否则编译程序段 2
形式 3：#ifndef 标识符
程序段
endif
作用：如果标识符没有被定义过，则编译程序段，否则不编译
形式 4：#ifndef 标识符
程序段 1
#else
程序段 2
endif
作用：如果标识符没有被定义过，则编译程序段 1，否则编译程序段 2

判断表达式，进行条件编译：
形式 1：#if(表达式)
程序段
endif
作用：如果表达式成立(表达式为“真”)则编译程序段，否则不编译
形式 2：#if(表达式)
程序段 1
#else
程序段 2
endif
作用：如果表达式成立则编译程序段 1，否则编译程序段 2

说明：①只对在判断范围内的程序段进行条件编译，其它的程序部分均编译
②不编译---不可能执行 ③省编译时间，减少目标程序的长度，便于程序移植

例：
```
#include"stdio.h"
#define Student 0
void main()
{
  #ifdef Student
```

```
        printf("处理学生信息...\n");
       {
         #if(Student)
            printf("      处理小学生信息...\n");
         #else
           printf("      处理大学生信息...\n");
         #endif
       }
     #else
       printf("处理教师信息...\n");
     #endif
}/*  改成：#define Student 1 再执行一次然后，把#define Student 1 用注释括起再执行一次  */
```

7 数组

数组：有序的同一类型的数据的集合

说明和规定：

①数组属于构造类型，数组中的所有元素必须属于同一类型

②数组元素的表示：数组名[数组下标]，例如：a[0],a[1],a[i],a[i+9]

下标：数组元素到数组起始位置的偏移量 p/120

③先说明后使用

④只能引用数组元素，而不能引用数组。例如：int j,a[100];

```
...
j=a[5]+a[0];
j=a;
```

⑤数组元素可以被赋值，也可以作为操作数出现在表达式中：a[0]=a[1+j]%j;

⑥定义数组时，元素的个数必须是整型常量或常量表达式。例如：char c[50]; char c[j];

⑦ C++ 不提供数组下标越界保护

⑧下标编号从 0 开始

7.1 数组定义和使用

1. 定义

格式：类型说明 数组名[数组长度表达式]

例：#define N 200;

```
float f[50];
int a[N*2];
```

说明：①数组长度即数组元素的个数，数组长度表达式中不能出现变量，表达式的值必须是正整型

②必须是方括号，不能是圆括号

2. 访问数组元素

数组元素作为操作数可以参加各类操作（表达式、函数调用）

注意：在使用时，方括号中可以是常量、含有变量的表达式，因为其值（必须是不超过长度的整数值）是数组下标。

例：int a[10],b[7];

```
a[0]=a[0]+b[7];
...
b[6]=max(a[7],b[0]);
b[7]=max(a[7],b[0]);
```

每个元素存放一个字符
凡是字符数组均可以用整型数组来实现

7.2 初始化数组

在定义数组的同时进行赋初值：
int a[10]={1,2,3};

说明：①数组定义时不初始化，各元素的值不确定
②初始值依次写在花括号中，之间用逗号分隔
③初始化时可以省略长度定义，系统自动取花括号内的数据个数作为数组长度
例如：int a[]={0,1,2,3,4}; 长度为 5
④定义数组时可以对前面的部分元素进行初始化，但是数组的长度不能省略
例如：int a[100]={11,22,33};
⑤初值可以是已赋过值的变量、含有赋过值的变量的表达式、常量表达式，例如：

```
#include"stdio.h"
void main()
{ int y=123,a[100]={y,y+5,5+55};
  printf("%d %d %d\n",a[0],a[1],a[2]);
} //执行结果：123 128 60
```

说明：如果 y 不赋初值，编译时会警告！

注意：不可以：int a[10]={1,,2,3}; p/125

7.3 字符数组

每个元素存放一个字符
凡是字符数组均可以用整型数组来实现

1. 字符数组定义：

char c0[10],c1[2][10];

2. 初始化

1. char a[10]={‘a’,’k’};
   即：a[0]=’a’; a[1]=’b’; a[2]～a[9]：没有初始化
   或者：char a[10]={“ab”};
   即：a[0]=’a’; a[1]=’b’; a[2]=’\0’; a[3]～a[9]：没有初始化
2. static char a[10]={‘a’,’b’};
   即：a[0]=’a’; a[1]=’b’; a[2]～a[9]：初始化成“空”
   数组 a 是静态变量
3. 缺省长度定义：
   char a[]={‘a’,’b’};
   系统自动判断数组 a 的长度为 2，即：a[2]
   char a[]={“ab”};
   系统自动判断数组 a 的长度为 3，即：a[3]；因为字符串有结束标志’\0’，由系统自动加上
4. 初始化的元素个数大于长度，则语法出错（编译通不过）：
   char a[2]={‘a’,’b’,’c’};
5. 对于二维数组，初始化的元素先分配给 0 行的各列，然后分配给 1 行的各列，…
   可以全部罗列，以逗号分隔；也可以按行进行初始化，各行用一对花括号括起，各行用逗号分隔
   例：char a[5][10]={‘1’,’2’,’3’,’4’,’5’,’6’,’7’};
   char a[5][10]={{‘a’,’b’},{‘r’,’o’,’&’},{‘*’,’+’}};
   注意：对于二维数组，低维长度不能省！高维长度可以省：
   char a[][3]={‘1’,’2’,’3’,’4’};
   系统自动认为：a[2][3] 的数组

3. 字符数组的使用

使用数组元素：可以被赋值，也可以作为表达式中的操作数、函数的实参
使用：数组名[下标]

4. 字符串和字符串结束标志：'\0'

在 C++ 中只有字符串常量，无字符串变量，对于可变的字符串，C++ 语言通过字符数组来实现并处理

字符串，字符串一定以'\0'作为结束标志：

例：char a[5]={'C','h','i','n','a'}; ----------无字符串结束标志
char a[6]={'C','h','i','n','a','\0'}; ----------有字符串结束标志
区别：如果以字符串的形式存放在数组 a 中，则 a 数组的长度至少为 6

字符串长度：从第 0 个元素（包括值为空格的元素）起，直至值为'\0'的之前的字符个数
即：字符串长度不包括字符'\0'，'\0'仅仅作为判断字符串是否结束用

```
例：#include"stdio.h"
    #include"string.h"
    void main()
    { char a[10];
      scanf("%s",a);
      printf("%s    l=%d\n",a,strlen(a));
    } /* 如果输入“abcd”，输出：abcd    l=4    */
```

如果用数组来存放字符串，则数组长度应定义成：字符串可能的最大实际长度+1

5. 字符数组的输入输出

'\0'只作判断用，不会输出！

C：利用输入输出函数以及格式符进行输入和输出：%c、%s

```
例：#include"stdio.h"
    void main()
    { char a[20];
      scanf("%s",a);
      printf("%s\n",a);  puts(a);
      getchar();
      gets(a);
      printf("\n\n%s\n",a);  puts(a);
      getchar();
    } /* 执行 3 次，分别输入字符串：“  123 456”↙
                                “123 456” ↙
                                “123” ↙,“    456  789” ↙    */
```

关于%s 字符串格式输入输出的说明：

①输出字符不包括'\0'，遇第一个'\0'结束
②输入时，以非空字符起，遇空格止，以“回车”为“写入”
③输入时，写数组名即可，可以不加地址符'&'(对于数组)
④输出时，写数组名而不是数组元素名(对于数组：用%s 格式输出)
⑤如果一次输入（以回车符为标志），自动以空格分隔，例：

```
char a[10],b[10],c[10],d[10];
scanf(“%s%s%s%s”,a,b,c,d);
```

如果输入：I am a student.
则：“I”→数组 a， “am” →数组 b， a→数组 c， student. →数组 d

⑥输入的字符个数超过数组的长度，取等于长度数的字符数，无字符串结束标志

C++：输入输出流对象 cin、cout

6. 有关处理字符串的函数的介绍

分别包含在“stdio.h”、“string.h”、“ctype.h”中 P/313

①字符串输出函数：puts(字符数组名)

例：char s[]={“China\nBeijing”}； // 花括号可省：char s[]=“China\nBeijing”;

puts(s);

输出：China

Beijing

注意：一次只能输出一个字符串，不能：puts(s1,s2);×

②字符串输入函数：gets(字符数组名)

说明：一次只能输入一个字符串

输入什么接受什么

输入超过长度，则根据长度自动截止；小于长度，后面的元素补’\0’

字符数组名前不必加地址符’&’，因为数组名本身就是该数组的地址

③字符串连接函数：strcat(字符数组名 1，字符数组名 2)

//strncat(char*,char*,int n)

作用：把字符数组 2 接在字符数组 1 的后面，然后赋值给字符数组 1；

该函数被调用结束，返回字符数组 1 的地址

说明：数组 1 的长度要足够大，如果长度不够，可以执行，但运算结果出错

原来数组 1 后面的’\0’自动取消，在连接后的字符串后自动加’\0’（如果长度够的话）

④字符串拷贝函数： strcpy(字符数组 1，字符数组 2 或字符串常量)

//strncpy(char*,char*,int n)

作用：把字符数组 2 或字符串常量拷贝至字符数组 1 中

例：strcpy(a1,a2);

strcpy(a,“asdfdfhjdskhg”);

注意：字符串不能直接进行赋值，例如：char a[10],b[10]={“123456789”};

a=b;×

数组 1 的长度必须 >= 数组 2 或字符串常量的长度

拷贝时，连同字符串结束符一起拷贝至字符串数组 1 中

例：
```
#include"stdio.h"
#include"string.h"
void main()
{ char a[10]={"abcdefghi"},b[5];
  printf("a: %s\nb: %s\n",a,b); /* 这时的数组 b 没有赋过值 */
  strcpy(b,a);
  printf("a: %s\nb: %s   l=%d     \n",a,b,strlen(b));
  printf("b[4]=%c    b[5]=%c   b[8]=%c\n",b[4],b[5],b[8]);
  getchar();
} /* 先执行该程序，然后把数组 b 的长度改成 10，再执行，比较有何区别 */
```

⑤字符串比较函数：strcmp(字符串常量 1 或字符数组名 1，字符串常量 2 或字符数组名 2)

//strncmp(char*,char*,int n)

作用：对两个字符串进行比较，然后返回比较结果：返回 0－－－两串相等

>0 的整数－－－串 1 > 串 2

<0 的整数－－－串 1 < 串 2

说明：• 从第 0 个元素起，逐个往后进行比较，自首次出现不同的字符就比较该字符的大小，后面的字符不予比较（不必长度一定要大）

例：”abcde”和”ac” ，串”ac”大

• 比较两个串的大小，只能用函数进行，不能：串 1==串 2

⑥求字符串长度的函数：strlen(字符数组名或字符串常量)

例：strlen(a); strlen(“gfhfjgjfgjdfg”);

说明：长度不包括字符串结束标志：’\0’

⑦大小写转换函数：strlwr(字符数组名或字符串常量) ：大写→小写

strupr(字符数组名或字符串常量) ：小写→大写

说明：串中的非字母字符不转换

注意：不同的系统所提供的库函数的函数名和功能有所不同！ 因为库函数本身不是 C 的组成部分。

7.4 向函数传递数组

如果实参是数组，则形参不管是指针形式还是数组形式，传递的都是数组的首地址。把实参（数组首地址）拷贝给形参（数组首地址），而不是把整个数组拷贝给形参。

数组元素、数组名都可以作函数的参数

其中：数组名既可以作形参、也可以作实参，进行整个数组的传递

而数组元素只能作实参，不能作形参(即，这时形参只能写成单变量形式，不能写成数组元素形式)

例：…

```
//error: int max(int x[],int y[]){ return ( x >= y ? x : y ); }
 int max(int x,int y){ return ( x >= y ? x : y ); }
    …
 void main()
 { int a[10]={1,2,3,4,5,6,7,8,9};
   cout<<max(a[7],a[9])<<" "<<max(a[8],a[0])<,endl;
 }
```

1. 数组元素作函数的参数

与变量一样，进行“值传递”

例：max_num=max(a[j],b[j]);

2. 数组名作函数参数

不是进行“值传递”，是把数组的首地址传递给被调用函数

数组名作参数时，调用函数和被调用函数的该参数都应说明成是数组，而且实参与形参都是数组名

例 1：求整数数组中的最大数：

```
#include"iostream.h"
#define N 10
void main()
{ int a[N]={1,2,3,4,5,66,7,8,9} ;
  int max(int a[],int);
  cout<<max(a,N);
}
int max(int a[],int n)
{ for(int m=0,i=0;i<n;++i)
    a[i]>m?m=a[i]:m;
  return m;
}
```

例 2：有 N 个学生，每个学生有 5 门课，计算各学生的平均成绩

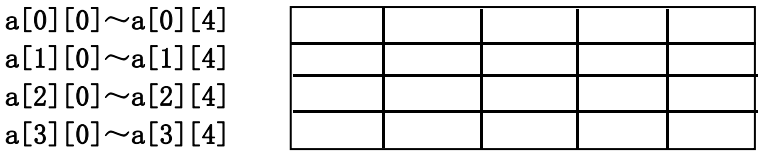
程序：

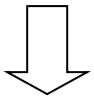
```
#include"iostream.h"
#define N 5
float average(float array[],int n)  //float average(float *array,int n)
{ int i;  float sum=0.0;
  for(i=0;i<n;++i)
    sum=sum+array[i];
  return(sum/n);
}
void main()
{ float array[N];  int i;
  cout<<"请输入该学生的"<<N<<"门课的成绩：\n";
  for(i=0;i<N;++i)
    cin>>array[i];
  cout<<"该学生的平均成绩："<<average(array,N)<<endl;
}
```

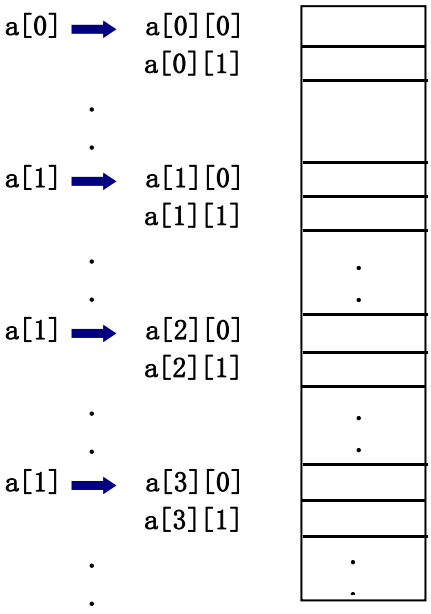
7.5 二维数组

a[4][5]:

a[0][0]～a[0][4]
a[1][0]～a[1][4]
a[2][0]～a[2][4]
a[3][0]～a[3][4]







1. 二维数组的定义：

类型说明 数组名[长度] [长度]
例：int a[10][10],b[100][2];
说明：①可以把二维数组看成是：一维数组中的每个元素是一维数组
②分配内存单元时：先分配行中的各列，然后再分配下一行的各列...

2. 二维数组的使用

元素表示形式：数组名[下标][下标]
例：a[0][0]、a[0][1]、a[5][5]
引用：a[1][2]=5;
b=a[0][0]*a[1][1]+c/x;

3. 二维数组的初始化

例：int a[3][4]={{1,2,3,4},{1}}, b[2][2]={0,1,2}, c[5][5]={{1},{2}};
说明：①初始化形式多样，但是总是从首元素开始分配
②按行进行初始化，每行用一对花括号括起，之间用逗号分隔
③整个按顺序进行初始化：一对花括号，无行的花括号
④部分元素初始化：自每行的首元素起被分配初值
⑤行维长度可省、列维长度不可省：
例：int a[][5]={0,1,2,3,4,5,6,7,8,9,10}; //省略第一维大小
共 11 个元素，系统自动判断行维长度为 3，
即：int a[3][5]= {0,1,2,3,4,5,6,7,8,9,10} ;
注意：不能：int a[3][4]={{1,2,3,4},{},{1}}

4. 降维处理

降维处理：在函数调用与被调用之际，把二维数组的首地址传递给形参，以达到把二维数组在被调用函

数内作为一维数组处理的目的

注意：数组首地址用首元素的地址，不要用数组名

5. 二维数组举例

例：把一个二维数组的行、列元素互换，并存到另一个数组中去

分析：如果是一个 2×3 的矩阵，则转换成 3×2 的矩阵

即：b[j][k]=a[k][j];

程序：

```
#include"iostream.h"
void main()
{ int a[2][4],b[4][2],j,k;
  cout<<"请依次输入 2×4 矩阵元素值，每行各元素间用逗号分隔，回车后另起一行：\n";
  for(j=0;j<2;++j)
    cin>>a[j][0]>>a[j][1]>>a[j][2]>>a[j][3];
  for(j=0;j<4;++j)
  { for(k=0;k<2;++k)
      { b[j][k]=a[k][j];
        cout<<b[j][k]<<"  ";
      }
    cout<<"\n";
  }
}
```

7.6 数组应用：排序

快速排序法：

```
#include"iostream.h"
void sort(int *a,int left,int right);
void main()
{ int a[100],max,i=0,j=0;
  cout<<"请输入整数序列，以-9999 结束输入，最多输入 100 个整数：\n";
  cin>>a[0];
  while(a[i]!=-9999)
  { i++; cin>>a[i]; }
  max=i-1;
  sort(a,0,max);
  for(i=0,j=0;i<=max;i++)
  { cout<<a[i]<<" "; j++;
  if(j>=10) { cout<<endl; j=0; }
  }
  cout<<endl;
}
void sort(int *a,int left,int right)
{ int pm,l=left,r=right,temp;
  pm=a[(left+right)/2];
  while(l<r)
  { while(a[l]<pm) ++l;
    while(a[r]>pm) --r;
    if(l>=r) break;
    temp=a[l];
    a[l]=a[r];
    a[r]=temp;
    ++l;  --r;
  }
  if(left<r)  sort(a,left,l-1);
  if(r<right) sort(a,r+1,right);
```

```
}  //    输入：1 2 3 9 0 4 10 6
     执行结果：0 1 2 6 3 4 9 10   */
```

跟踪执行：

```
#include"iostream.h"
void sort(int *a,int left,int right)
{ int pm,l=left,r=right,temp;
  pm=a[(left+right)/2];
  while(l<r)
  { while(a[l]<pm) ++l;
    while(a[r]>pm) --r;
    if(l>=r) break;
    temp=a[l];
    a[l]=a[r];
    a[r]=temp;
    ++l;  --r;
  }
  int i=0,j=0;
  while(a[i]!=-9999)
    { cout<<a[i]<<" "; i++; j++;
      if(j>=10) { cout<<endl; j=0; }
    }cout<<endl;
  if(left<r)  sort(a,left,l-1);
  if(r<right) sort(a,r+1,right);
}
void main()
{ int a[100],max,i=0,j=0;
  cout<<"请输入整数序列，以-9999 结束输入，最多输入 100 个整数：\n";
  cin>>a[0];
  while(a[i]!=-9999)
    { i++; cin>>a[i]; }
  max=i-1;
  sort(a,0,max);
}//输入：1 2 3 9 0 4 10 6  分析执行结果 */
```

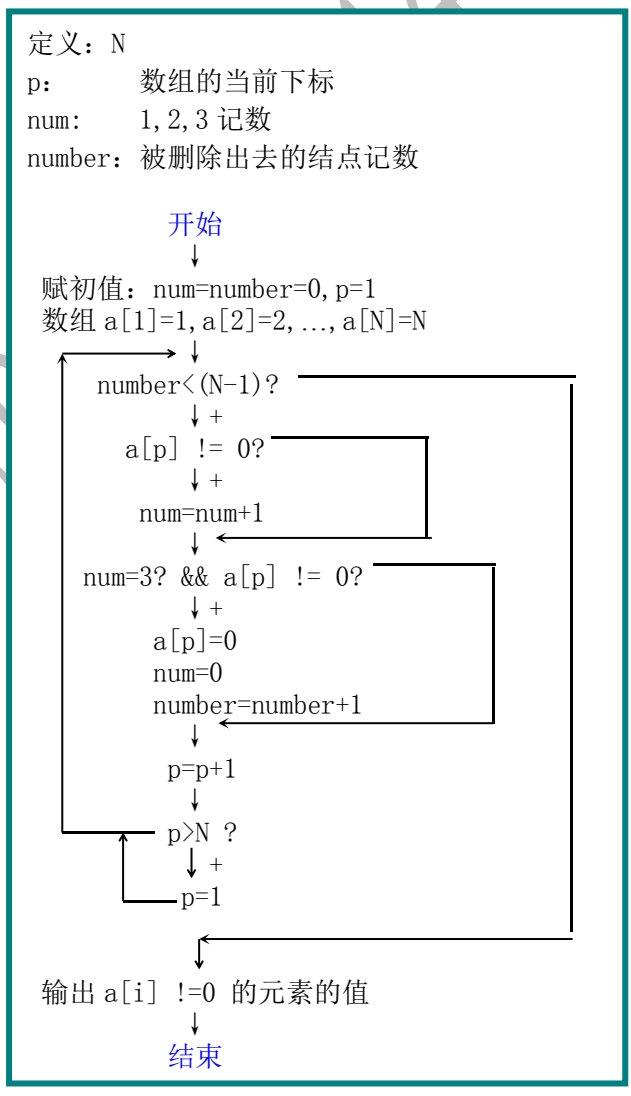
正确程序：

```
#include"iostream.h"
void sort(int *a,int left,int right)
{ int pm,m,l=left,r=right,temp;
  m=(left+right)/2; pm=a[m];
  while(l<r)
  { while(a[l]<pm) ++l;
    while(a[r]>pm) --r;
    if(l>=r) break;
    temp=a[l];     //互换
    a[l]=a[r];
    a[r]=temp;
    //++l;  --r;  改成如下 if 语句：
    if(l==m && r!=m) ++l;
    else if(l!=m && r==m) --r;
    else { ++l; --r; }
  }
  int i=0,j=0;
  if(left<r)  sort(a,left,l-1);
  if(r<right) sort(a,r+1,right);
}
```

```
void main()
{ int a[100],max,i=0;
  cout<<"请输入整数序列，以-9999 结束输入，最多输入 100 个整数：\n";
  cin>>a[0];
  while(a[i]!=-9999)
    { i++; cin>>a[i]; }
  max=i-1;
  sort(a,0,max);
  i=0;
  while(a[i]!=-9999)
    { cout<<a[i]<<" "; i++; }
  cout<<endl;
}//输入：1 2 3 9 0 4 10 6  分析执行结果 */
```

7.7 数组应用：Josephus 问题

/* N 个人排成一圈，从编号为 1 的人开始数 1、2、3，数到 3 的人被开除出圈，再不断地数，直至剩下一个人，问编号为几？ */



程序：

```
#include"iostream.h"
#define N 7
void main()
{ int num=0,p=1,a[N+1],i,number=0;
  for(i=0;i<=N;++i)
      a[i]=i;
  while(number<N-1)
    { if(a[p]!=0)
        ++num;
      if(num==3 && a[p]!=0)
        { a[p]=0; num=0; number++; }
      ++p;
      if(p>N)
        p=1;
    }
  for(i=1;i<=N;++i)
  { if(a[i]!=0)
    { cout<<"最后一个获胜者的编号是:
"<<i<<endl; break; }
  }
}
```

8 指针

在 C++中：任何变量都有地址，这些变量都可以用指针来指向并进行操作

8.1 指针概念

1. 指针类型

每个数据类型都有相应的指针

指针有类型，什么类型的指针只能指向该类型的对象，不能指向其他不同类型的对象

2. 指针变量的定义

```
int x,y,*px,*py;
char c1,c2[10],*pc;
```

注意：在变量定义语句中，每定义一个指针，在指针变量名前必须加’*’

3. 建立指针（初始化指针）

操作符 & ：取址

```
int x,y,*px=&x,*py=&;
char c1,c2[10],*pc1=&c1,*pc2=c2;
```

4. 间接引用指针变量

操作符 * ：取指针变量所指向的单元内的内容

例：

```
#include"iostream.h"
void main()
{ int x=100,y=x+200,*p=&y;
  char a[10]="abcdefg",*pc=a;
  cout<<y<<" "<<*p<<" "<<p<<endl;
  cout<<a<<" "<<*pc<<" "<<pc<<endl;
}    /* 300 300 地址值
        abcdefg a abcdefg  */
```

说明：指针变量的间接引用可以作左值

5. 指针变量的地址

指针本身是变量，所以指针变量也有地址，指针的地址是"指针的指针"

例：

```
#include"iostream.h"
void main()
{ int x=100,*p=&x,**pp=&p;
  cout<<x<<" "<<*p<<" "<<*pp<<" "<<**pp<<endl;
}  /* 100 100 0x0066FDF4 100  */
```

6. 定义指针和使用指针的区别

定义语句中的'*' 和'&'：

```
int x=123,*p=&x,y=*p;    //int x=123,*p=&x,y=x;
```

例：

```
#include"iostream.h"
void main()
{ int x=123,*p=&x,y=*p;          //定义指针 p 和使用指针 p(等价：int x=123,*p=&x,y=x;)
  cout<<x<<" "<<*p<<" "<<y<<endl;//使用指针 p
}   /* 123 123 123  */
```

7. 指针的初始化

即：给指针变量一个初始指向 p/149

8. 指针类型与实际存储的匹配

强调：指针所指向的变量的类型必须与指针类型一致

8.2 指针运算

指针变量描述的是地址，是无符号量，不同于整数和无符号整数

指针间的算术运算（指针相加、相乘、相除、模）均无意义！

但是，在一定的条件下，指针可进行：

1. 两个指针变量的比较运算
   ① 当指针变量 p1 和 p2 指向某一变量时，可以进行如下逻辑判断：
   p1==p2 ——— 如果相等，则 p1 和 p2 指向同一变量
   p1!=p2 ——— 如果不等，则 p1 和 p2 指向的不是同一变量
   ② 当指针变量 p1 和 p2 指向同一个数组变量时，可以进行如下逻辑判断：
   p1==p2 ——— 如果相等，则表示指向数组的同一元素
   p1!=p2 ——— 如果不等，则表示指向的不是同一元素
   p1>p2 ——— 如果成立，则表示 p1 指向的数组元素在 p2 指向的数组元素之后
2. 两个指针变量的减法运算（两个指针变量的加法无意义）
   当两个指针 p1 和 p2 指向同一数组时，可进行减法运算：
   |p1-p2| ——— 绝对值表示 p1 指向的对象（数组元素）与 p2 指向的对象之间的元素个数
3. 一个指针变量与整数的运算
   当指针指向数组时，可进行加、减一个整数的运算：
   p±n ：从当前指向的数组元素上移或下移 n 个元素

8.3 指针与数组

实参是数组，形参不管是指针形式还是数组形式，传递的都是数组的首地址。把实参（数组首地址）拷贝给形参（数组首地址），而不是把整个数组拷贝给形参。

在 C++中：任何可以用数组下标完成的操作，都可以用指针来实现
使用指针数组可以使程序更紧凑、灵活
数组的指针：数组的地址
数组元素的指针：数组元素的地址
指向数组的指针变量：用变量表示一个指向数组的指针
对于任何一个已经定义过的指针变量，即可以指向同类型的任一变量，又可以指向同一类型的一维数组；当指向数组时，便是数组指针，可以指向该数组中的任一元素

1. 指向一维数组元素的指针

(1) 定义数组指针

```
int a[10],*p=a;（或：int a[10],*p=&a[0]; 不可以：int a[10],*p=&a;）
或者：int a[10],*p;
      p=a;
      …
      注意：不能写成：*p=a;    *p=&a[0];
            可以：    p=a; *p=*a; p=&a[0]; *p=a[0];
```

(2) 数组名与数组指针变量

数组名是数组的起始地址，也是数组的第 0 个元素的地址
所以：如果指针变量 p 指向数组 a：int a[10],*p=a;
① *p、*a、a[0]：等价
② p+i
   a+i
   } 均表示 a[i]元素的地址，指向元素 a[i]
③ a[i]、p[i]
   *(p+i)
   *(a+i)
   } 均表示 a[i]的值，就是 a[i]
④ p 可以改变值，数组名不可以；即：指针变量的值可变，数组名值不可变，因为数组名本身是数组变量、而不是指针变量
例：p++; 合法：表示当前指针值加 1，指向下一个元素
a++; 非法

(3) 数组元素的引用

如果 a 是数组名，p 是指向 a 的指针

下标法引用：a[j] 形式
指针法引用：*(a+j)　*(p+j)　p[j]

```
例：#include"iostream.h"
    void main()
    { int a[10]={123,11,22,33,44,55,66,77,88,99},*p=a,i;
      p=a+8;          // p 指向 a[8]
      p--;            // p 指向 a[7]
      p=p-2;          // p 指向 a[5]
      *p=a[3];        // a[5]=a[3]=33
      *++p=600;       // *(++p)=600;  p 指向 a[6], a[6]=600
      *(a+3)=300;     // a[3]=300
      for(i=0;i<10;i++)
        cout<<"a["<<i<<"]= "<<a[i]<<"  ";
    }
```

2. 数组名、数组指针变量作函数参数

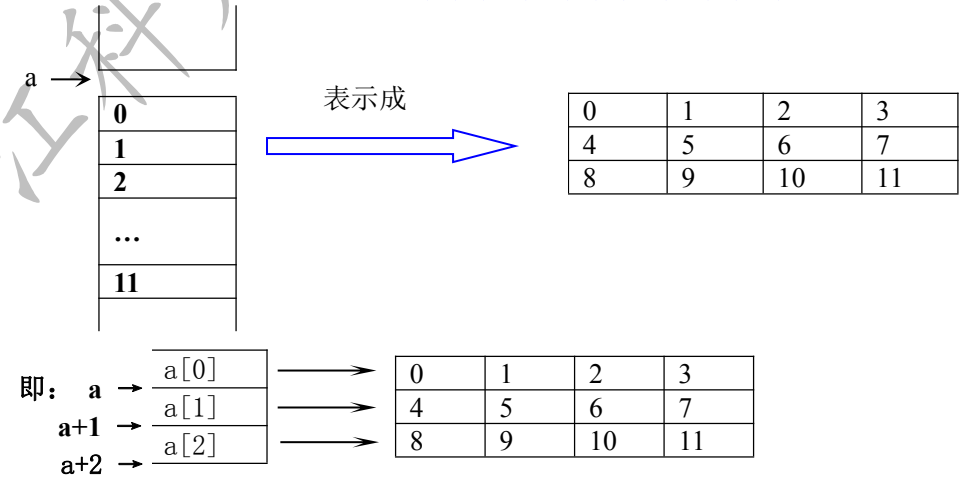
1. 数组名和数组指针变量都是地址，作为函数参数均传递地址
   数组名：传递的是首地址
   数组指针变量：传递的是当前地址
2. 不管实参是数组名，还是指针变量，形参都可以定义成数组名或指针变量
   操作时要注意：数组元素不可越界

3. 指向二维数组的指针

(1) 二维数组元素的地址

对于二维数组 a:
形式 1：a　———　数组首地址，第 0 行首地址：元素 a[0][0]的地址
　　a+1 ———　第 1 行首地址：元素 a[1][0]的地址
　　a+2 ———　第 2 行首地址：元素 a[2][0]的地址
　　…
形式 2：可以把 a[0]、a[1]、a[2]、…，看成一维数组名，所以：
　　a[0] ———　第 0 行首地址：元素 a[0][0]的地址
　　a[1] ———　第 1 行首地址：元素 a[1][0]的地址
　　a[2] ———　第 2 行首地址：元素 a[2][0]的地址
　　…
形式 3：a[i]+j ———　表示第 i 行第 j 列的地址：元素 a[i][j]的地址

如果定义二维数组 int a[3][4]={{0,1,2,3},{4,5,6,7},{8,9,10,11}}：



所以：① 数组名本身是指针，但不可以改变值

② 如果用二维数组名表示地址：

*a　　a[0]　　*(a+0)　　&a[0][0] ———等价

a[i]　　*(a+i)　　&a[i][0]　　———等价

a[i]+j　*(a+i)+j　　&a[i][j]　　———等价

③ 对于二维数组元素：

a[i][j]　　*(a[i]+j)　　(*(a+i))[j]　　*(*(a+i)+j) ———等价

注意：① 二维数组名 a 并不表示元素 a[0][0]的地址，而是数组 a 的地址，数组 a 的地址就是 a[0]的地址，

即是第 0 行的地址；

所以：a 和 *a 不等价

② a[i]中是元素 a[i][0]的地址，而(a+i)表示 a[i]的地址，所以：a+i 和 a[i]不等价

(2) 二维数组的指针

注意：不能把一维数组指针用于二维数组

理解成：由 n 个元素（n 为二维数组的列数）所组成的数组的指针

例：int a[3][4],(*p)[4]=a;　/* (*p)[4]的圆括号不能缺，否则为指针数组 */

说明：① 指针值可以改变，表示指针的指向改变，可以进行有意义的运算：p±n

② 表示地址：

*p　　p[0]　　*(p+0) ——— 等价：&a[0][0]

p[i]　　*(p+i)　　——————— 等价：&a[i][0]

p[i]+j　*(p+i)+j ——————— 等价：&a[i][j]

③ 间接访问：

(*(p+i))[j]　　*(p[i]+j)　　*(*(p+i)+j) ——— 等价：元素 a[i][j]的值

注意：对二维数组指针 p，*p 是 0 行的首地址：p[0]；*(p+i)是 i 行的首地址：p[i]

例 1：
```
#include"iostream.h"
void main()
{ int a[3][4]={{1,3,5,7},{9,11,13,15},{17,19,21,23}};
  int (*p)[4]=a,i,j;
  for(i=0;i<3;i++)
       { for(j=0;j<4;++j)
           cout<<*(p[i]+j)<<" ";
         cout<<endl;
       }
  cout<<"\n\n";
  for(i=0;i<3;i++)
       { for(j=0;j<4;++j)
           cout<<(*(p+i))[j]<<" ";
         cout<<"\n";
       }
} /* 执行结果：1 3 5 7
                9 11 13 15
                17 19 21 23

                1 3 5 7
                9 11 13 15
                17 19 21 23    */
```

例 2：
```
#include"iostream.h"
void main()
{  int a[6][2]={{1,3},{5,7},{9,11},{13,15},{17,19},{21,23}};
   int (*p)[2],i;
   p=a+1;       /* 二维数组指针 p 指向二维数组 a 的第 1 行首地址 */
   for(i=0;i<5;p=p+1,i++)
```

```
            cout<<*p[0]<<" ";
        cout<<"\n\n";
        for(p=a;p<a+6;p++)  /*  p 指向二维数组 a 的第 0 行首地址，输出数组 */
            cout<<**p<<"   "<<*(*(p)+1)<<endl;
    }/* 执行结果：5 9 13 17 21
                  1     3
                  5     7
                  9     11
                  13    15
                  17    19
                  21    23   */
```

4. 二维数组指针的降维处理

```
例：#include"stdio.h"
    void print(int* p,int n) //一维数组指针 p
    { for(int i=0;i<n;++i)
        printf("%d: %d\n",i,p[i]);
    }
    void main()
    { int a[2][3]={1,2,3,4,5,6};
      print(&a[0][0],6);    //二维数组 a 的第一个元素的地址作函数的实参
    }
```

8.4 堆内存分配　　p/155

动态内存分配，释放动态内存分配

C：malloc(size) ， calloc(n,size)) ， free(指针标识符)

堆：允许程序在运行时申请某个大小的内存空间

堆内存：动态内存

new、delete 操作符：

C++ 专用操作符，不必用头文件声明

```
例：#include "iostream.h"
    #define N 100
    void main()
    { int *p1,*p2;
    p1=new int[N];
    p2=new int;
    …
    delete []p1;   //delete []p1,p2;
    delete p2;
    }
```

const 指针

constant

关键字：const

作用：冻结，修饰类型描述符，得到该类型的派生类型

被 const 的对象不能再被更新，所以在定义时必须初始化

```
例：const int a=555,b=1;  //等价于 int const a=555,b=1;
    a=777;            //出错
    b=666;            //出错
```

1. 指向常量的指针

指向常量的指针：指针指向的变量值不能变，指针本身的值可变

即：对于指向常量的指针 p，*p 不可以再被赋值

例：const int a=78,b=27;//a,b 都是常量，定义时必须初始化
```
const int *p=&a;     //指向常量的指针，指针指向的变量值不能变，指针本身的值可变
                     //或：const int a=78,b=27,*p=&a;
int c=17;
a=98;              //出错
*p=97;             //出错
p=&b;
*p=999;            //出错
p=&c;
*p=98;             //出错：尽管 c 不是 const 的，但是规定 p 指向的变量值不能变
c=98;
```

注意：对于指向常量的指针 p，*p 不可以作左值

2. 指针常量

定义时，在指针名前加 const，标识指针本身是常量，不可以改变指针的指向。但是，指针所指向的变量的值可以改变。

```
例：#include "iostream.h"
    void main()
    { int a,*const p1=&a,b,*p2=&b;  //为提高程序可读性，一般不在同一行上进行这样的声明
      a=100;  b=200;
      cout<<*p1<<" "<<*p2<<"    ";
      cout<<a<<" "<<b<<" "<<endl;
      *p1=222;
      //p1=&b;  //出错：不可以改变指向
      p2=&a;
      cout<<*p1<<" "<<*p2<<"    ";
      cout<<a<<" "<<b<<" "<<endl;
    }/* 执行结果：100  200      100  200
                 222  222      222  200  */
```

注意：①定义指针常量时必须初始化，因为指针值不能被改变
②对于指针常量 p，p 不可以作左值

3. 指向常量的指针常量

```
例：const int a=7;
    int b;
    const int *const p1=&a,*const p2=&b;
    p1=&b;   //出错：p1 是指向常量的指针常量
    *p1=39;  //出错：p1 是指向常量的指针常量
    *p2=39;  //出错：p2 是指向常量的指针常量，尽管 b 不是 const 常量
    b=39;
    p2=&a;   //出错：p2 是指向常量的指针常量
    a=39;    //出错：a 是 const 常量
```

注意：①定义指向常量的指针常量时必须初始化，因为指针值不能被改变
②对于指向常量的指针常量 p，p 不可以作左值，*p 也不可以作左值

8.5 指针与函数

1. 传递数组的指针

数组名是指针，但不是指针变量，所以对数组名不可以进行赋值运算

如果把数组名传递给形参，则形参就是一个指针，所以在形参所在的函数内，这个指针可以改变值

例：#include"iostream.h"
```
int max(int a[],int n)    //或者：int max(int *a,int n)
{ int m=0;
  for(int i=0;i<n;i++)
  if(a[i]>m)  m=a[i];
  a=a+3;    //ok
   cout<<*a<<"  ";
   return m;
}
void main()
{ int a[10]={111,222,333,444,555,666,777,888,999,0};
  //a=a+3;          //error
  cout<<max(a,10)<<endl;
}/* 执行结果：  444  999  */
```

2. 使用指针修改函数参数

指针变量作函数的参数：地址传递，不是变量的值传递
一次函数调用仅返回一个值，如果要在被调用函数的中改变多个值，利用指针作为函数的参数

例：每次输入 2 个数 a 和 b，如果 a<b，则 a、b 互换并输出
程序：
```
#include"iostream.h"
void main()
{ void swap(int *,int *);
  int a,b,*pa=&a,*pb=&b;
  char c='y';
  while(c=='y'||c=='Y')
    { cout<<"\n\n 请输入 a，b ：";
      cin>>a>>b;
      if(a<b)
        swap(pa,pb);     //或者：swap(&a,&b);
      cout<<"a= "<<a<<" b= "<<b<<endl;
        cout<<"    继续( y/n ) ？";
      cin>>c;
    }
}
void swap(int *a,int *b)
{ int m;
  m=*a,*a=*b,*b=m;
}
```

注意：① 用指针（变量的地址：&a）或指针变量作函数实参，函数的形参必须定义成指针型
② 指针变量作参数，实参、形参均为同类型的指指针型
是单向地址传递，所以调用函数可以改变实参指针变量所指向的变量的值，而不改变实参指针变量本身的值（即：变量的地址不会被改变）

3. 指针函数

指针函数：返回指针的函数称指针函数
即：函数的返回值是一个指针（是一个地址）

函数说明格式：

类型说明 *函数名(参数列表)

例：在给定的字符串 s 中查找某个字符 c，如找到、返回在 s 中第一次出现的位置，并把自此字符起的余串输出，如找不到、则返回 NULL（空指针）

```
程序：#include"iostream.h"
      void main()
      { char *strfind(char *,char c);
        char s[20],c,*p;
        cout<<"请输入一个字符串：";
        cin>>s;
        cout<<"\n 请输入一个被查找的字符：";
        cin>>c;
        if((p=strfind(s,c))!=NULL)
             cout<<"位置："<<p-s+1<<"  余串："<<p<<endl;
        else
             cout<<"串："<<s<<" 中无字符: "<<c<<"\n";
      }
      char *strfind(char *s,char c)
      { char *p=NULL;   //p 是局部变量
        do{ if(*s==c)
              { p=s;  break; }
          }while(*(++s)!='\0');
        return p;        //VC++6.0 不警告
      }
```

4. void 指针

空类型指针：只存放地址，不能进行指针运算，不能进行间接引用

```
例：#include"iostream.h"
    void main()
    { void *p;
      int x=500,y;
      char c='?',cc[10]="abcdefgh";
      p=&x;
      //y=*p;                          //error: 不能进行间接引用
      y=*(int *)p;                     //强制进行类型转换
      cout<<y<<endl;
      p=&c;
      //cout<<*p<<endl;                //error: 不能进行间接引用
      cout<<(*(char *)p)<<endl;
      p=cc;  cout<<(*(char *)p)<<endl;
      //cout<<(*(p+1))<<endl;          //2 个 error：不能进行指针运算，不能进行间接引用
      //cout<<(*(char *)(p+1))<<endl;  //error：不能进行指针运算
    }/*  执行结果：500
                   ?
                   a        */
```

8.6 字符指针

C++ 语言中：字符指针、字符数组、字符串常量一起构成 C++ 语言中灵活的字符串处理的基本功能

对字符串处理：C++ 编译程序安排一静态存储区域，并自动给字符串未加'\0'，实际所占空间比串的实际长度大 1

字符串指针：字符串的首地址

指向字符串的指针变量：用变量表示所指向的字符串的地址

1. 字符数组和字符串常量

字符串：字符数组，字符串常量；字符串以 '\0'结束
字符数组：是变量，每个元素存放一个字符
字符串常量：用双引号括起的字符串，其类型：char *

例如：char a[10]="123456789";　　//a 是字符数组，"123456789"是字符串、但不是字符串常量
　　cout<<"123456789"<<endl;　　//"123456789"是字符串常量
　　cout<<a<<endl;

2. 字符串常量的格式和特点

字符串常量存放在数据区的"常量段"中
例：与 p/168 的例题结果不同(环境：VC++ 6.0)

```
#include"iostream.h"
void main()
{ char *p1="abcd",*p2="abcd";//如果 *p2="abcdef", 则结果是：no equal
  if(p1==p2)
      cout<<"equal \n";
  else
      cout<<"no equal \n";
  if("join"=="join")          //如果 "join"=="joinn", 则结果是：no equal
      cout<<"equal \n";
  else
      cout<<"no equal \n";
  char a1[10]="12345",a2[10]="12345";
  if(a1==a2)
      cout<<"equal \n";
  else
      cout<<"no equal \n";
}/* 执行结果：  equal
               equal
               no equal    */
```

3. 字符指针

```
例：char *p,*cp="ShangHai";     /* 指向字符串常量的首地址 */
    …
    p= "I am a student.";
    …
```

所以：*cp　　———表示字符串的首字符
　　*(cp+i) ———表示字符串的第 i 个字符

凡是形参是字符数组或字符指针，均可以用字符串的指针作实参，进行地址传递，只是注意下标不要越界

```
例：#include"iostream.h"
    void main()
    { char *p2="abcd";
      cout<<*p2<<" "<<*(p2+2)<<" "<<p2<<endl;
    }//执行结果：  a c abcd
```

4. 有关处理字符串的函数的介绍

包含在“string.h”头文件中
①字符串连接函数：strcat(字符数组名 1，字符数组名 2)
　作用：把字符数组 2 接在字符数组 1 的后面，然后赋值给字符数组 1；

该函数被调用结束，返回字符数组 1 的地址

说明：数组 1 的长度要足够大，如果长度不够，可以执行，但运算结果出错

原来数组 1 后面的'\0'自动取消，在连接后的字符串后自动加'\0'（如果长度够的话）

②字符串拷贝函数： strcpy(字符数组 1，字符数组 2 或字符串常量)

作用：把字符数组 2 或字符串常量拷贝至字符数组 1 中

例：strcpy(a1,a2);

strcpy(a,“asdfdfhjdskhg”);

注意：字符串不能直接进行赋值，例如：char a[10],b[10]={“123456789”};

a=b;×

数组 1 的长度必须 >= 数组 2 或字符串常量的长度

拷贝时，连同字符串结束符一起拷贝至字符串数组 1 中

例：
```
#include"stdio.h"
#include"string.h"
void main()
{ char a[10]={"abcdefghi"},b[5];
  printf("a: %s\nb: %s\n",a,b); /* 这时的数组 b 没有赋过值 */
  strcpy(b,a);
  printf("a: %s\nb: %s   l=%d     \n",a,b,strlen(b));
  printf("b[4]=%c    b[5]=%c   b[8]=%c\n",b[4],b[5],b[8]);
  getchar();
} /* 先执行该程序，然后把数组 b 的长度改成 10，再执行，比较有何区别 */
```

③字符串比较函数：strcmp(字符串常量 1 或字符数组名 1，字符串常量 2 或字符数组名 2)

作用：对两个字符串进行比较，然后返回比较结果：返回 0－－－两串相等

>0 的整数－－－串 1 > 串 2

<0 的整数－－－串 1 < 串 2

说明：• 从第 0 个元素起，逐个往后进行比较，自首次出现不同的字符就比较该字符的大小，后面的字符不予比较（不必长度一定要大）

例：”abcde”和”ac”，串”ac”大

• 比较两个串的大小，只能用函数进行，不能：串 1==串 2

④求字符串长度的函数：strlen(字符数组名或字符串常量)

例：strlen(a); strlen(“gfhfjgjfgjdfg”);

说明：长度不包括字符串结束标志：'\0'

⑤大小写转换函数：strlwr(字符数组名或字符串常量) ：大写→小写

strupr(字符数组名或字符串常量) ：小写→大写

说明：串中的非字母字符不转换

注意：不同的系统所提供的库函数的函数名和功能有所不同！ 因为库函数本身不是 C++ 的组成部分。

5. 字符串赋值

例：
```
#include"iostream.h"
void main()
{ char *p2="abcd",a[10]="7890";
  cout<<*p2<<" "<<*(p2+2)<<" "<<p2<<endl;
  cout<<*a<<" "<<*(a+2)<<" "<<a<<endl;
  p2="1234";            // 可以
  cout<<*p2<<" "<<*(p2+2)<<" "<<p2<<endl;
  //a="jkliu";          //error：不可以
}/* 执行结果：  a c abcd
               7 9 7890
               1 3 1234        */
```

注意：字符数组只能通过字符串拷贝函数进行变值

8.7 指针数组

指针数组：是变量，是用指向同一数据类型的指针组成的数组变量
即：数组中的每个元素都是指针变量，且都是指向同一数据类型的指针

1. 定义指针数组

定义：类型标识符 *数组名[整型常量表达式]；
例：int *a[10]； ---有 10 个元素 a[0]～a[9]，每个元素都是整型指针变量
char *c[100]； ---有 100 个元素 c[0]～c[99]，每个元素都是字符型指针变量
注意：int (*a)[10]；是二维数组的指针
说明：① 每个元素都是指针变量
② a 是指针数组的名，不能对 a 进行增量运算
a 是 a[0]的地址，a+i 是 a[i]的地址
*a *(a+i)分别表示：a[0]和 a[i]的值
③ 用途：主要用于处理长度不一的字符串
用二维数组处理长度不一的正文效率低，而指针数组由于每个元素都是指针变量，所以通过地址来操作正文行就很方便
④ 指针数组的每个元素只能存放地址
例：char *a[3]={"111111","22222222","33333"}; // a 是指针数组

2. 指针数组与二维数组的区别

字符指针数组：每个元素是一个指针，每个指针所指向的字符串长度可以不等
二维字符数组：每一行所包含的字符个数相同

3. 指向指针的指针

即：指向指针的指针变量
格式：类型 **指针名；
例如：int x,*p=&x,**p1=&p;
即：定义变量 p 是一个指向整型变量的指针，定义 p1 是一个指向“整型变量的指针”的指针
也就是：p1 中存放的是整型指针 p 的地址，而 p 中存放的是整型变量的地址

```
例：#include"iostream.h"
    void main(int i,char *s[])
    { int x=12345,*p=&x,**p1=&p;
      cout<<x<<" "<<*p<<" "<<**p1<<endl;
    } /*  执行结果：12345  12345  12345  */
```

4. NULL 指针

NULL 指针：空指针，即一个没有“指向”的指针，也即没有初始值的指针
void * 指针：无类型指针
例：void *p1; //p1 是无类型的指针
int *p2=NULL; //p2 是有类型的空指针

8.8 命令行参数

1. 命令行参数

如果可执行文件（命令行）需要带参数时（即：程序需要接受命令行参数），可以把主函数定义成：

```
void main(int argc,char *argv[])   // argument count ,  argument vector
{ ...
  ... }
```

其中：argc ---命令行参数的数目，包括命令本身
argv ---指向参数的字符指针数组

例：回打参数

```
程序：#include"iostream.h"
      void main(int num,char *s[])
      { int j;
        for(j=1;j<num;j++)
          cout<<s[j]<<" ";
        cout<<endl;
      }
      /* 在 DOS 下发命令：  cgy hello name1 name2 name3 ok↙
         结果为：          name1 name2 name3 ok
         如果发命令：      hello cgy↙
         则执行结果为输出：cgy                          */
```

2. main() 函数的返回

main() 函数返回到系统：return 1; //必须有函数返回类型：int main()
或：exit(1); //必须包含头文件：stdlib.h

8.9 函数指针

在 C++ 语言中，函数本身不是变量，但是与数组名类似：函数有地址
函数指针：指出函数地址的指针
通过函数指针可以调用相应的函数
因此：① 允许用指针指向函数
② 函数有类型，因此指向函数的指针也有类型的区别

1. 函数指针的定义

格式：类型说明 (*指针变量名)(形参说明列表);
例：int (*pf)(); pf 为指向返回整型值的函数的指针

2. 函数指针的内在区别

省略()的函数是地址：max()是函数，max 是函数 max()的地址
函数指针的类型：是函数的返回值的类型
函数指针的类型与同类型的数据指针不能相互指向，也不能进行显式类型转换

3. 通过函数指针调用函数

```
例：#include"iostream.h"
    void ff1(int a,int b)
      { cout<<(a*a+b*b)<<endl; }
    void ff2(int a,int b)
      { cout<<(a*a)<<endl; }
    void main()
    { void (*p)(int,int); /* 定义指针 p 是指向有两个整型参数、无返回值的函数的指针*/
      p=ff1;
      p(2,5);
      (*p)(2,5);
      p=ff2;
      p(2,5);
      (*p)(2,5);
    }/*  执行结果：29
                   29
                    4
                    4  */
```

4. 用 typedef 简化函数指针

```
例：typedef int (*FunP)(int,int);  //声明 FunP 是整型类型、且有 2 个整型参数的函数的指针
    FunP fp;
```

例：把上面的程序改成如下，执行结果相同：

```
typedef void (*FunP)(int,int);  //声明 FunP 是整型类型、且有 2 个整型参数的函数的指针
#include"iostream.h"
void ff1(int a,int b)
  { cout<<(a*a+b*b)<<endl; }
void ff2(int a,int b)
  { cout<<(a*a)<<endl; }
void main()
{ FunP p;
  p=ff1;
  p(2,5);
  (*p)(2,5);
  p=ff2;
  p(2,5);
  (*p)(2,5);
}/*  执行结果：29
              29
               4
               4   */
```

5. 函数指针作函数参数

例：对任意输入的两个数：或输出大的数，或输出小的数，或输出两个数的和(取决于命令行的参数)的程序如下。

```
#include"iostream.h"

void ff(int a,int b,int (*p)(int,int))
            /* 定义指针 p 是指向有两个整型参数、返回整型值的函数的指针*/
{ int x;
  x=(*p)(a,b);  /* 或：x=p(a,b); */
  cout<<x<<endl;
}

void main(int i,char *s[])
{ int a,b,min(int,int),max(int,int),sum(int,int);
  if(i!=2)
     cout<<"命令行的参数输入出错!\n";
  else
    { cout<<"请输入两个整数：";
      cin>>a>>b;
      if(*s[1]=='1')      ff(a,b,max);
      else if(*s[1]=='2') ff(a,b,min);
      else                ff(a,b,sum);
    }
}

int min(int a,int b)
{ return(a>b?b:a); }

int max(int a,int b)
{ return(a>b?a:b); }

int sum(int a,int b)
{ return(a+b); }
```

```
/* 如果该程序的可执行文件名为：cgy.exe 则：
   执行时输入：>cgy 1↙    就调用 max()函数
   执行时输入：>cgy 2↙    就调用 min()函数
   执行时输入：>cgy 3↙    就调用 sum()函数  */
```

6. 函数指针数组

7. 返回函数指针的函数

即：函数的返回值是一个指针（是一个地址）

函数说明格式：

函数指针类型说明 *函数名(参数列表)

9 引用

C++ 中有 2 种间接访问变量的途径：指针、引用

9.1 引用的概念

引用：给对象起“别名”，只能对“左值表达式”进行引用

引用运算符：&

注意：①&在定义时出现在赋值运算符的左边表示是“引用”，否则是取址符

②一个对象一旦有了别名，此别名就不能再作为别的对象的别名，所以声明时必须进行初始化

③有了别名的对象，不管对“真名”、还是对“别名”进行操作，都是对此对象进行操作。

④一个被声明成是引用的变量并不占存储空间,仅与另一个占有存储空间的变量间建立了一种“映射”

9.2 引用的声明和使用

1. 声明

语法：被引用对象的类型标识符 &引用标识符=左值表达式

```
例：int i,a[100];
    int &ii=i,&aa=a[10]; //ii 是变量 i 的“引用”， aa 是数组元素 a[10]的“引用”
```

2. 使用操作

使用一个对象的引用，如同使用其“真名”

```
例：#include "iostream.h"
    void main()
    { int x=100;
      int& xx=x,y=999,&yy=y;  //为提高程序可读性，一般不在同一行上进行这样的声明，应分行进行
      cout<<x<<" "<<xx<<endl;
      cout<<y<<" "<<yy<<endl;
    }/* 执行结果：  100 100
                   999 999 */
```

9.3 什么可以被引用

对于除 void 外的类型 T，如果一个变量被声明为 T& 时，则这个变量必须能够用 T 类型的一个变量的值，或者能够用一个可以转换成 T 类型的对象进行初始化。

符合上述条件的对象，都可以声明引用（即：可以起一个或多个别名）

```
const double &rr=1;  //合法，(在 VC++6.0 中，double &rr=1;非法)
```

指针也是变量，可以对指针变量进行引用：

例：#include "iostream.h"
```
void main()
{ int a=12345,*p=&a,*&pp=p;
  cout<<a<<" "<<*p<<" "<<*pp<<endl;
}/* 执行结果： 12345 12345 12345 */
```

注意：①有空指针，无空引用。所以对 void 型指针引用非法（void 仅仅在语法上相当于一个类型）
②数组名本身不是一个变量的标识符，所以不能对数组名进行引用
③T&是类型 T 的引用，T&本身不是一种数据类型，所以不能对“引用”进行引用，也不可以有引用的指针
④引用是对变量的引用，而不是对类型的引用

9.4 用引用传递函数参数

即：用引用作函数的参数

例：#include "iostream.h"
```
void swap(int &x,int &y)
{ int j;
  j=x;
  x=y;
  y=j;
}
void main()
{ int a=12345,b=54321;
  cout<<a<<" "<<b<<endl;
  swap(a,b);
  cout<<a<<" "<<b<<endl;
}/* 执行结果： 12345 54321
                54321 12345 */
```

优点：起到指针的作用，但并不建立参数的拷贝

注意：实参必须是“左值表达式”，例如实参不可以：a+5

引用所存在的问题： **p/191**

9.5 返回多个值

由于引用是“别名”，实参与形参之间不是值传递，而是一种“映射”，所以对形参的改变，实际上就是对实参的改变。

9.6 用引用返回值

即：对函数的返回值进行引用

p/193 的例题分析：
```
#include "iostream.h"
float temp;
float fn1(float r)
{ temp=r*r*3.14;
  return temp;
}
float & fn2(float r)
{ temp=r*r*3.14;
  return temp;
}
void main()
```

```
{ float a=fn1(5.0);
  //float &b=fn1(5.0);  //error: 函数 fn1 的返回类型是 float，不是 float&
  float c=fn2(5.0);
  float &d=fn2(5.0);
  cout<<a<<"  "<<c<<"  "<<d<<endl;
}/* 执行结果：  78.5  78.5  78.5  */
```

如果把变量 temp 设置成局部的：

```
#include "iostream.h"
//float temp;
float fn1(float r)
{ float temp;
  temp=r*r*3.14;
  return temp;
}
float & fn2(float r)
{ float temp;
  temp=r*r*3.14;
  return temp;    //wrning: 返回一个局部变量的地址
}
void main()
{ float a=fn1(5.0);
  //float &b=fn1(5.0);  //error: float 与 float& 之间的类型不能转换
  float c=fn2(5.0);
  float &d=fn2(5.0); //危险：d 引用到一个局部变量上
  cout<<a<<"  "<<c<<"  "<<d<<endl;
 }/* 执行结果：  78.5  78.5  78.5  */
```

9.7 函数调用作为左值

例：

```
#include "iostream.h"
    int & fn(int index,int a[])
    { int &r=a[index];//如果：int r=a[index]; 则下一句警告：返回一个局部变量的地址
      return r;       //返回: a[index]
    }
    void main()
    { int a[]={1,3,5,7,9};
      cout<<(fn(2,a)=55)<<" "<<a[2]<<endl;   //输出：右→左
      cout<<a[2]<<endl;
    }/* 执行结果：  55  5
                    55     */
```

说明：函数 fn()的返回值是一个引用，即引用到变量 r 上；而 r 是 a[index]的引用，所以实际上函数的返回值“引用”到变量 a[index]上。所以，函数调用 fn()可以作为左值表达式。

注意：

①并不是所有的函数返回值都可以被引用的，例如：

```
    int & fn(int index,int a[])
    { int r=a[index];
      return r;       //返回: a[index]
    }
```

这里的 r 不是 a[index]的引用，r 并不与一个本函数外的某个对象有联系，所以当本函数结束时就释放 r，返回 r 并不能起到"引用"的作用（编译时警告：返回一个局部变量的地址。留下潜险）

②引用返回值的目的：对函数的返回值进行赋值，作为一种程序设计的技巧

9.8 用 const 限定引用

问题的提出：

调用函数时，要建立实参的副本（把实参拷贝给形参），如果被传递的数据类型很大，开销也很大，不可取。如果采用指针或引用的形式，则可以不建立副本，但是会产生实参被修改的危险（如果不希望实参被修改的话）。

所以，如果调用函数时不建立副本以提高运行效率，又不希望有实参被改动的危险，则采用传递 const 指针或 const 引用（即：实参是地址或引用，形参定义成 const 型的指针或引用）的方法。

p/198 的例题讲解

说明：

```
例：#include "iostream.h"
    void main()
    { const double &a=1; //const double const &a=1; 将警告
      cout<<a<<endl;
    }/* 执行结果： 1 */
```

由于“引用”a 本身不能被重新赋值，所以“引用”总是 const 的，因此不必对引用 a 再用关键字 const 加以限定

9.9 返回堆中变量的引用

“引用”本身不是指针，也不能作指针，所以不能直接对“堆”进行引用。

```
例：int &p=new int(2); //非法
```

合法：

```
#include "iostream.h"
void main()
{ int *p=new int(202),&pp=*p;
  cout<<pp<<endl;
  delete &pp;          //或者：delete p;  释放堆空间
}/* 执行结果： 202 */
```

10 结构

10.1 结构

1. 结构的基本概念

数据类型：基本类型、指针类型、构造类型

构造类型：数组、结构体、共用体

前面介绍的数据都是只包含一种类型信息，数组是构造类型，但是只能存放同一种类型的数据

结构体、共用体可以把不同类型的数据对象组织在一起，而形成一种新的类型的变量，这种变量属于构造类型，由用户自己进行构造，以处理一些复杂的数据

例如，对于学生，每个学生都有：学号、姓名、性别、年龄、出生日期、班级、宿舍信息；这些信息各属于不同的数据类型，不能用数组对 N 个学生的信息进行处理；希望一个学生的信息组成一条记录，N 个学生 N 条记录，便于管理和处理

利用结构体和共用体实现

结构体与共用体的区别：

结构体：内存所占长度为各成员所占内存长度之和

即：各个成员占有自己的内存单元

共用体：内存所占长度为成员中最长的成员的长度

即：各个成员共用内存单元

2. 定义结构类型和结构变量

必须先定义结构体类型，然后才可以定义该类型的变量

(1) 定义结构类型

一般格式：

```
struct 结构名
{ 结构体成员 1 定义;
  结构体成员 2 定义;
   …
  结构体成员 n 定义;
};
```

例：通讯录：姓名－－－字符串
地址－－－字符串
电话－－－无符号长整型
邮政编码－无符号长整型

定义：

```
struct address
{ char *name;
  char *add;
  unsigned long tel;
  unsigned long post_code;
};
```

或者：

```
struct address
{ char name[10];
  char add[30];
  unsigned long tel;
  unsigned long post_code;
};
```

即：用户自己定义了一个新的类型：**address** 类型，该类型属于结构体的数据结构

注意：定义一个结构仅仅声明了一个新的数据类型，系统对结构类型不分配内存

(2) 结构变量的定义

一旦定义了结构类型，就可以定义该类型的变量－－－结构变量

形式 1：先定义结构类型，再定义结构变量

```
例：struct address
    { char *name;
      char *add;
      unsigned long tel;
      unsigned long post_code;
    };
    …
    struct address add1,add2;
```

形式 2：定义结构类型的同时，定义结构变量

```
例：struct address
    { char *name;
      char *add;
      unsigned long tel;
      unsigned long post_code;
    }add1,add2;
```

方法 3：直接定义，但无结构类型名，只有结构变量名（无名结构）

```
例：struct
    { char *name;
      char *add;
      unsigned long tel;
      unsigned long post_code;
```

```
}add1,add2;
```

说明：① 用户定义了一个数据类型：struct address
可以理解为：在 int，char，float 等类型外，又增加了一种数据类型
② address 是一个 struct 的类型，而不是变量名！
不能对 address 进行赋值等运算，而只能用来定义该类型的变量！
③ 对 address 类型不分配空间，只对其变量分配空间
④ 结构中的某一成员可以又是另一个结构类型的变量
⑤ 某一成员的名可以与结构外的其它变量同名
⑥ 可以嵌套定义结构类型，但是在某个结构中定义的结构类型只在该结构中有效

3. 访问结构成员

(1) 结构体成员运算符：.

优先级：最高
结合性：左→右

(2) 结构体变量的成员

例：add1.name
add1.tel
add2.nae
add2.post_code

(3) 引用

结构体成员与单个变量一样，可以进行各种合法的运算、赋值等操作，但是必须指出其所属的变量名
例如：add1.post_code=200030;

```
例：struct date
      { int month;
        int day;
        int year;
      };
    struct student
      { int number;
        char name[10];
        char sex;
        int age;
        struct date brithday;
        char add[30];
      }student1,s99[101];
      …
      student1.number=12345;
      student1.brithday.day=29;
      student1.brithday.month=7;
      student1.brithday.year=1980;
      s99[0].number=1;
      s99[7].brithday.day=5;
      …
      …
      student1.brithday.year++;
      student1=s99[77];
```

4. 结构初始化和赋值

(1) 初始化

在定义结构体变量的同时赋初值，不能在结构体类型的定义中赋初值

例：#include"stdio.h"

```
struct student
{ int number;
  char name[10];
  char sex;
  int age;
  struct date{ int year,month,day; }brithday;   //内嵌定义另一个结构类型和对象
  char add[30];
}student1={9801,"余雨",'m',18,1980,12,1,"华山路 1954 号"}; //初始化

或者：struct student student1={9801,"余雨",'m',18,1980,12,1,"华山路 1954 号"};
```

(2) 赋值

由于结构大小确定、成员次序固定，所以，可以把一个结构体变量直接赋值给另一个同类型的结构体变量

上例中的：student1=s99[77];

10.2 结构与指针

指向结构体变量的起始地址

结构体变量在内存中的存放：

按各成员在结构体类型定义中的次序，顺序存放

1. 结构体变量的指针的定义

```
...
struct student student1,student2,*p=&student1;
```

2. 指向结构体成员的运算符：—>

优先级：与运算符“．”同

结合性： 左→右

当一个指针 p 定义为某个结构体类型的指针后，该指针可以指向任何同类型的变量；并且可以通过指针间接访问成员

访问结构体的成员：

直接访问：student1.name

间接访问：p=&student1; p->name

(*p)：表示 p 所指向的那个结构体变量

注意：括号不能少

例如：如果 p=&student1;

则：student1.name
(*p).name
p->name
} 等价：表示结构体变量 student1 的成员 name 的值

注意：① 结构体变量的指针用于指向整个变量，不能指向某个成员：p=&student1.age;×

② 对于单个结构体变量的指针 p：p++ 和 p=p+1 p--均无意义

③ ++p->age; 等同于 ++(p->age); 即年龄加 1

(++p)->age; 无意义

p->age++; 等同于 (p->age)++;

10.3 结构与数组

上面例题中的变量 s99[101] 就是一个有 101 个 student 结构体类型的元素的数组

```
例：#include"iostream.h"
    #define N 3
    void main()
    { float h_c1=0,h_c2=0,h_c3=0,h_p=0;
      int num_c1,num_c2,num_c3,num_p;
      struct student{ int num;          /* 学号 */
                      char name[10];    /* 姓名 */
                      float c1,c2,c3,p; /* 3 门课程及其平均成绩 */
                     }s[N];  /* 定义 N 个同学的结构体数组 */
      cout<<"请依次输入各个同学的学号，姓名，3 门课程的成绩(用空格分隔)：\n";
      for(int i=0;i<N;++i)
      cin>>s[i].num>>s[i].name>>s[i].c1>>s[i].c2>>s[i].c3;
      cout<<"\n\n 各个同学的平均分统计如下：\n";
      for(i=0;i<N;++i)
        { s[i].p=(s[i].c1+s[i].c2+s[i].c3)/3;
            cout<<s[i].num<<" "<<s[i].name<<" "<<s[i].p<<endl;
          if(s[i].c1>h_c1)
            { h_c1=s[i].c1;  num_c1=i; }
          if(s[i].c2>h_c2)
            { h_c2=s[i].c2;  num_c2=i; }
          if(s[i].c3>h_c3)
            { h_c3=s[i].c3;  num_c3=i; }
          if(s[i].p>h_p)
            { h_p=s[i].p;  num_p=i; }
        }
      cout<<"\n 课程 c1 最高分的同学信息：\n";
      cout<<s[num_c1].num<<"  "<<s[num_c1].name<<"  "<<s[num_c1].c1<<"  "<<s[num_c1].c2<<"
"<<\
           s[num_c1].c3<<" 平均分:  "<<s[num_c1].p<<endl;
      cout<<"\n 课程 c2 最高分的同学信息：\n";
      cout<<s[num_c2].num<<"  "<<s[num_c2].name<<"  "<<s[num_c2].c1<<"  "<<s[num_c2].c2<<"
"<<\
           s[num_c2].c3<<" 平均分:  "<<s[num_c2].p<<endl;
      cout<<"\n 课程 c3 最高分的同学信息：\n";
      cout<<s[num_c3].num<<"  "<<s[num_c3].name<<"  "<<s[num_c3].c1<<"  "<<s[num_c3].c2<<"
"<<\
           s[num_c3].c3<<" 平均分:  "<<s[num_c3].p<<endl;
      cout<<"\n3 门课程平均最高分的同学信息：\n";
      cout<<s[num_p].num<<" "<<s[num_p].name<<" "<<s[num_p].c1<<" "<<s[num_p].c2<<" "<<\
           s[num_p].c3<<" 平均分:  "<<s[num_p].p<<endl;
    }
```

10.4 传递结构参数

即：把结构遍历作为实参传递给形参

如果形参是实参的"引用"，则实参与形参之间不进行"拷贝"

p/212～214 例题讲解

10.5 返回结构

一个函数的返回值可以是一个结构变量，如果返回的是结构的"引用"或是结构指针，则注意被返回的结构变量不能是局部变量。p/214～217

10.6 链表结构

链表：由 n 个数据成员组成，每个成员看作是链的一个结点
每个成员都是同一类型的数据，可以是任何合法的类型之一
每个结点的信息除数据之外，还包含“下一个结点”的地址
一个“链头指针”：存放链的起始地址
链的最后一个结点的“下一个结点”的地址为“NULL”：表示链结束
一个“链尾指针”：存放链的最后一个结点的地址（便于 append 一个结点）

单向链表
双向链表：再增加一个信息：“上一个结点”的地址（便于删除一个结点）

所以：链表的成员的数据类型是一个结构体，该结构体类型的成员之一必定是指向该结构体的指针

例如：
```
struct student{ int num;
                char name[10];
                float c1,c2,c3,p;
                struct student *p_next;
              };
```

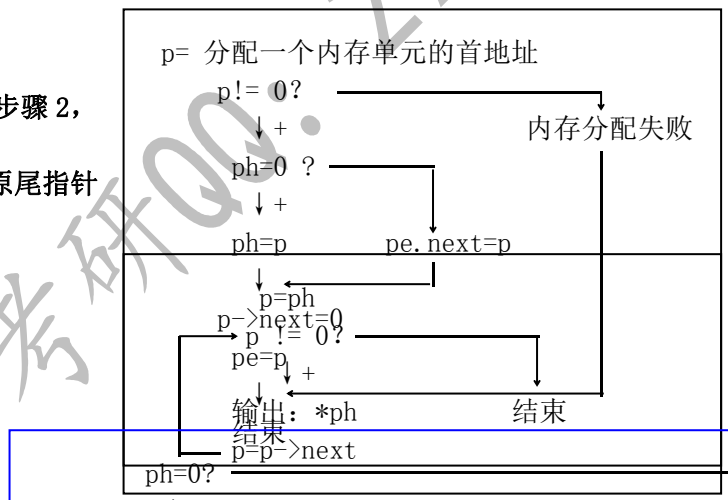
10.7 创建与遍历链表

1. 建立链表

新建链表的过程：
链头指针：ph=0
链尾指针：pe＝0

然后在链表中加入结点（ append ）：

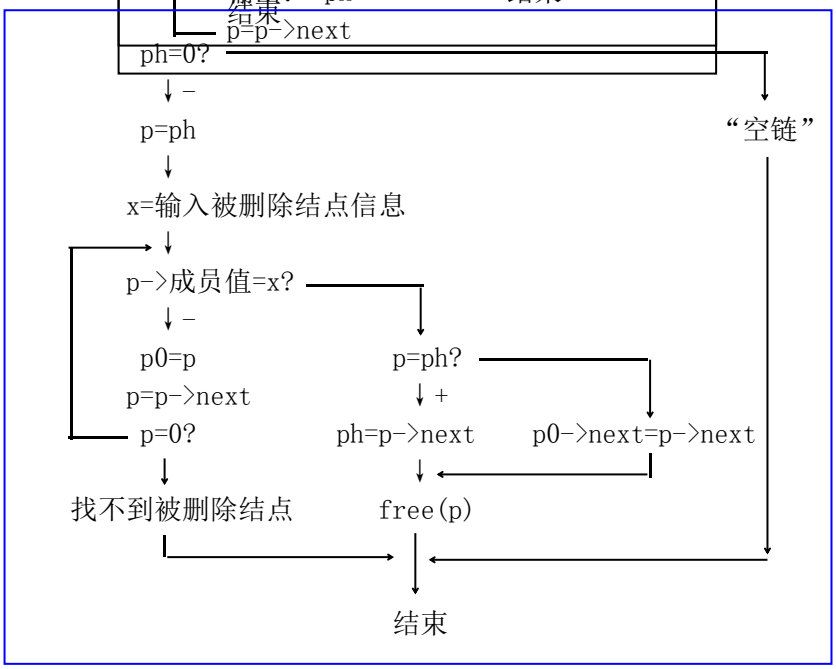
1. 分配内存单元，地址 p；成功执行步骤 2，否则输出出错信息后执行步骤 4
2. 是空链？是空链：头指针 ph=p,否则原尾指针 pe 的下一个成员地址是 p
3. p 是链尾：p->next=0, pe=p
4. 结束



2. 遍历与输出链表

输出链表的过程：
链头指针：ph
链尾指针：pe

1. p=ph，执行 2
2. p=0?(是空链？)，是执行 4；否则执行 3
3. 输出：*p，p=p->next，重复步骤 2
4. 结束



10.8 删除链表节点

删除结点：
p:当前节点指针，p0:前一节点指针

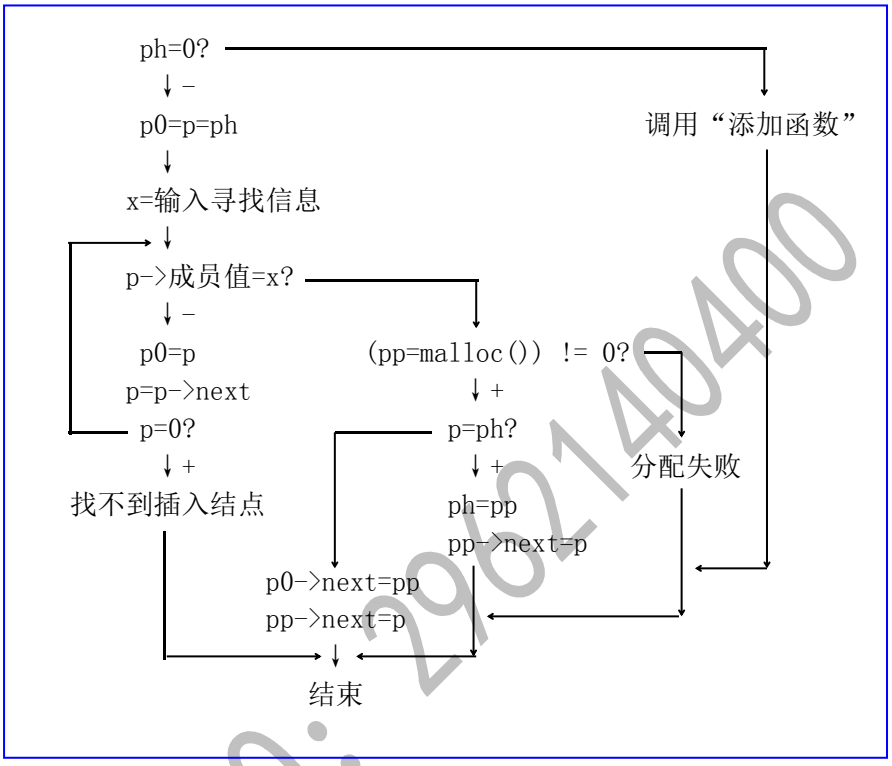
1. Ph=0?(空链？)，是输出“空链”，执行 8；否则 p=ph，执行 2
2. 输入被删除结点的信息
3. p->成员值 = 输入信息？，是执行 5，否则执行 4
4. p0=p，p=p->next，p=0?(链结束？)，是输出“找不到被删除结点”，然后执行 8，否则重复步骤 3
5. 删除该节点：p=ph?(链首结点)，是，ph=p->next，然后执行 7；否则执行 6
6. p0->next=p->next

7．释放 p 所指向的内存单元
8．结束

10.9 插入链表节点

插入结点：

1．Ph=0?(空链？)，调用“添加结点函数”(append())，调用毕执行 9；否则 p=ph，p0=ph(p0=插入的定位结点的前一个结点的地址)，执行 2
2．输入插入点信息
3．p->成员值 = 输入信息？，是执行 5，否则执行 4
4．p0=p，p=p->next，p=0?(链结束？)，是输出“找不到插入结点”，然后执行 8，否则重复步骤 3
5．pp=新分配内存单元的地址，pp!=0?(分配成功？)，是输出“分配失败”，然后执行 8，否则执行 6
6．p=ph?(链首结点)，是：ph=pp，pp->next=p，然后执行 8；否则执行 7
7．p0->next=pp，pp->next=p
8．结束



10.10 结构应用

1. 例题 1

猫抓了 n（n>1）个老鼠后宣布：老鼠排队，自此先后次序不变。猫每天吃掉单数位置的老鼠，最后剩下一只小老鼠，问这只小老鼠的首次编号是几？

分析：用链，每个结点包含两个信息：首次编号，下一结点地址
先建立 n 个结点的链，然后从链中删除“奇”结点，直至一个结点止

程序：

```
#include"iostream.h"
struct mouse{ int n;
              struct mouse *next;
            };
int cat_mouse(int num)
{ struct mouse *ph,*pe,*p,*p0;  /* ph:链首指针，pe:链尾指针 */
  if(ph=new(struct mouse))
    { ph->n=1; pe=ph; pe->next=NULL;
          for(int i=2;i<=num;++i)                    /* 建链 */
        { if(p=new(mouse))      //在 C++中 struct mouse 的 struct 可以不写，但 C 中必须写
            { p->n=i;
              p->next=NULL;
              pe->next=p;
              pe=p;
            }
          else
            { cout<<"内存单元分配失败！\n";
```

```
            return 0;
          }
      }
        do{ p=ph=ph->next; p0=NULL;
        for(i=2;p!=NULL;p=p->next,++i)
          { if(i%2==0)
              p0=p;          /* p0：前一结点的地址 */
            else             /* 奇结点：删除 */
              p0->next=p->next;
          }
      }while(ph->next!=NULL); /* 删除"奇"结点 */
        return(ph->n);
    }
  else
    { cout<<"内存单元分配失败！\n";
        return 0;
    }
}
void main()
{ int num;
  cout<<"请输入老鼠只数( >1 ): ";
  cin>>num;
  cout<<"最后剩下的一只老鼠的编号是:
      "<<cat_mouse(num)<<endl;
}
```

2. Josephus 问题

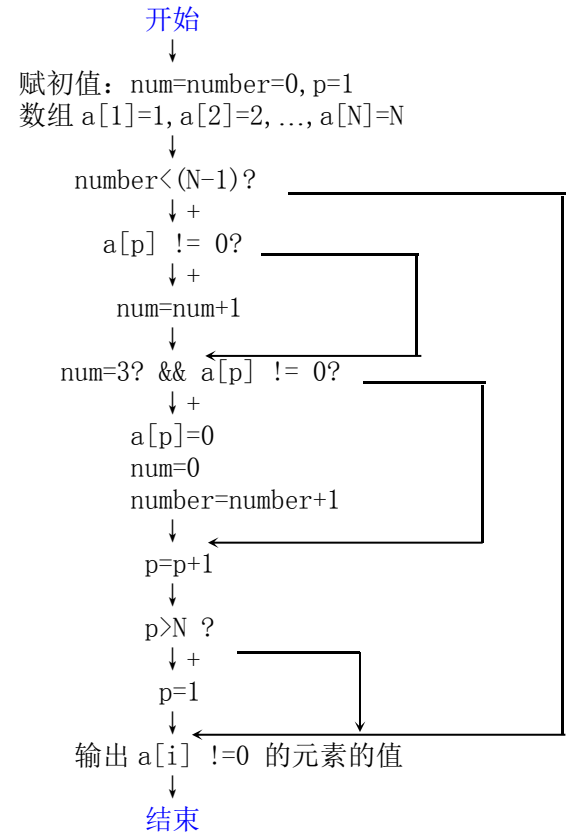
p/226～228

N 个人排成一圈，从编号为 1 的人开始数 1、2、3，数到 3 的人被开除出圈，再不断地数，直至剩下一个人，问编号为几？

方法 1：用数组（7.7 节中的方法）

```
#include"iostream.h"
#define N 7
void main()
{ int num=0,p=1,a[N+1],i,number=0;
  for(i=0;i<=N;++i)
      a[i]=i;
  while(number<N-1)
    { if(a[p]!=0)
        ++num;
      if(num==3 && a[p]!=0)
        { a[p]=0; num=0; number++; }
      ++p;
      if(p>N)
        p=1;
    }
  for(i=1;i<=N;++i)
  { if(a[i]!=0)
    { cout<<"最后一个获胜者的编号是："<<i<<endl; break; }
  }
}
```

定义：N
p:　　数组的当前下标
num:　　1,2,3 记数
number：被删除出去的结点数



方法 2：用结构

```
#include"iostream.h"
struct Josephus{ int ordinal;
                 Josephus *next;
               };
void main()
{ int number,num,n=0;
  Josephus *ph,*pl,*pr,*pdel;
  cout<<"请输入总小孩数、数小孩的间隔数(用空格分隔)：";
  cin>>number>>num;
  ph=pl=new(Josephus);ph->ordinal=1; //创建第一个节点，并初始化
  for(int i=2;i<=number;++i)          //创建第 2～第 number 个节点、初始化
     { pr=new(Josephus); pr->ordinal=i; pl->next=pr; pl=pr; }
  pr->next=ph;                        //构成一个"环"
  pl=pr=ph;
  while(pr->next!=pr)  //如果 pr->next!=pr ： 只剩一个节点
    { ++n;
      if(n==num)       //如果 n=num ： 该节点从环中剔除
        { pl->next=pr->next;
          pl=pr->next;
          pdel=pr;
          pr=pr->next;
          delete pdel;
          n=0;    //再从头开始数数
        }
      else
        { pl=pr; pr=pr->next; }
    }
  cout<<"最后一个获胜者的编号是："<<pr->ordinal<<endl;
}
```

11类 p/232

类：是 C++ 程序设计的核心成分，是 C++ 封装的基本单元

11.1 从结构到类

1. 从结构到类

在 C 中，结构体不允许有成员函数
在 C++ 中，结构体允许有成员函数。
类与结构体的区别：
除关键字不同外（class，struct）的唯一区别是，结构在默认情况下的成员是公共的，而类在默认情况下的成员是私有的。

```
例：#include "iostream.h"
    void main()
    { struct A{ int x,y;
                void show()
                { cout<<"x="<<x<<"  y="<<y<<endl; }
              };
      A a;
      a.x=3; a.y=9;
      a.show();
    }/* 执行结果：  x=3  y=9  */
```

2. 类的构成与定义

类可以由成员构成，成员可以是数据成员、或成员函数。
成员可以是“私有”(private）的、“公有”(public）的、或“保护”(protected）的。

```
类的定义：class 类名标识符{ private:
                              …
                              …
                              protected:
                              …
                              …
                              public:
                              …
                              …
                            };
```

私有成员：只允许类本身的其他成员访问，任何本类以外的成员（对象）均不能访问。本类以外的对象只能通过该类的公有成员函数对私有成员进行访问。
保护成员：基类私有成员中需提供给派生类访问的那部分的成员，应定义成保护成员
公有成员：提供的外部接口，允许类的可使用者访问

```
例：class Circle{ private:
                    double center_x,center_y,radius;
                  public:
                    void make_circle(double x,double y,double r);
                    void make_circle(double x,double y,double r);
                };
```

说明：①各段次序不要求固定
②private、protected、public 为访问控制修饰符（关键字），修饰在它们之后列出的成员被程序的其他部分所使用的权限
③private 段如果放在最前面，关键字 private 可省，否则不可省；protected、public 段不论出现在何处均不可以省
④不同类的成员可以同名

11.2 软件方法的发展必然

结构化程序设计 ⟶ 面向对象程序设计

结构化：程序 = 算法 + 数据结构
面向对象：程序 = 对象 + 对象 + 对象 +…
对象 =算法 + 数据结构

11.3 定义成员函数

1. 成员函数的命名

成员函数命名： 类标识符::成员函数标识符

:: 作用域区分符

例：Circle::make_circle(double x,double y,double r);
即，类 Circle 的成员函数：make_circle(double x,double y,double r)

说明：不同作用域的对象可以同名，在一个作用域内使用可以使用的同名对象时应加域区分符（::）以区分

p/237 的例题说明

2. 成员函数的定义

函数声明
函数定义

对类的每项操作都是通过成员函数实现的，使用某个操作就发生一次函数调用

⑴常规定义
即：在类内进行声明（说明），在类外进行定义
语法格式：返回类型 类名::成员函数名(形参列表)
{ …
…
}

```
例：void Circle::make_circle(double x,double y,double r)
{ crnter_x=x;
  center_y=y;
  radius=r;
}
```

⑵内联成员函数定义
两种形式：显式、隐式

显式：加关键字 inline

```
例：#include "iostream.h"
class Circle{ private:
                    double center_x,center_y,radius;
              public:
                    inline void make_circle(double x,double y,double r);
                    inline void show_circle();
            };
inline void Circle::make_circle(double x,double y,double r)
{ center_x=x;
  center_y=y;
  radius=r;
}
inline void Circle::show_circle()
{ cout<<"x="<<center_x<<"  y="<<center_y<<"  radius="<<radius<<endl; }

void main()
{ Circle c1,c2;
  c1.make_circle(1.,5.,7.);
  c2.make_circle(11.,55.,77.);
  c1.show_circle();
  c2.show_circle();
}/* 执行结果：  x=1  y=5  radius=7
               x=11  y=55  radius=77  */
```

隐式：全部在类内进行定义

```
例：#include "iostream.h"
class Circle{ private:
                  double center_x,center_y,radius;
              public:
                  void make_circle(double x,double y,double r)
```

```
                { center_x=x;
                      center_y=y;
                      radius=r;
                }
               void Circle::show_circle()
               { cout<<"x="<<center_x<<"  y="<<center_y<<"  radius="<<radius<<endl; }
          };

void main()
{ Circle c1,c2;
  c1.make_circle(1.,5.,7.);
  c2.make_circle(11.,55.,77.);
  c1.show_circle();
  c2.show_circle();
}/* 执行结果：  x=1  y=5  radius=7
              x=11  y=55  radius=77  */
```

说明：对于开发大型的软件，通常把类的定义和其成员函数的定义分开进行。

3. 重载成员函数

同一个类的成员函数可以重载，方法与非成员函数重载相同
不同类的成员函数同名，不是函数重载
类的成员函数与非成员函数同名，不是函数重载

例：p /239 ～ 240

11.4 调用成员函数

即：成员函数的使用

1. 类与对象

类：是一种共享机制，提供了同类对象的共同的操作的实现
在 C++中，把具有相同的内部存储结构和具有相同的一组操作的对象看成是同一类的

类与对象的关系： int j,k; //int 是类，j,k 是 int 类的 2 个对象
定义一个类，并不占数据成员的存储空间；定义了一个类的对象，该对象占有数据成员的存储空间

类的成员（数据成员、或成员函数）必须通过类的对象进行访问（只能访问公有成员）
类的成员函数必须通过类的对象进行调用

(1) 对象的定义

与结构体类似

```
例：class Circle{ private:
                    double center_x,center_y,radius;
                public:
                    int k;
                    void make_circle(double x,double y,double r)
                     { center_x=x;
                       center_y=y;
                       radius=r;
                     }
                    void Circle::show_circle()
                              {      cout<<"x="<<center_x<<"             y="<<center_y<<"
```

```
radius="<<radius<<endl; }
                }c1,c2,*pc=&c1;
```

或者：Circle c1,c2,*pc=&c1;

(2) 访问类的成员

操作符：. 和 ->

例：c1.make_circle(1,2,3); pc->make_circle(1,2,3);

```
c2.make_circle(7,8,9);
c1.k=999; c2.k=777;
```

2. 调用成员函数

(1) 调用成员函数的方法

必须通过具体的对象进行调用

p /240 例题分析

(2) 通过指针调用成员函数

pc -> make_circle(1,2,3); p /241

(3) 利用“引用”访问成员函数

把形参定义成是实参的“引用”

p /242 例题

(4) 在成员函数中访问成员

在成员函数中访问本类的成员，直接使用即可（只要本成员函数中不屏蔽掉类的成员）

p /243 例题

3. this 指针

this 指针是 C++ 实现封装性的一种机制

程序运行时，系统自动为每个类提供一个 this 指针，一个类的当前对象的地址由该 this 指针指出，所有对成员的访问都被隐含地加上前缀：this-> （p/243 的最下面举例）

4. 类的定义和使用举例

例：编写一个处理学生类的程序，具有输入学生信息、输出学生信息的功能

程序：

```
#include "stdio.h"
#include "iostream.h"

class Student{ private:
                int number,age;
                char name[10],sex[3];
              public:
                void new_s(void); //如果函数没有参数，可以在括号中写 void
                inline void view_s();
             };

void Student::new_s()
{ cout<<endl<<"please input: number name age sex :"<<endl;
  cin>>number>>name>>age>>sex;
}
inline void Student::view_s()
{ cout<<"display: "<<endl;
  cout<<"         number: "<<number<<endl;
  cout<<"         name: "<<name<<endl;
  cout<<"         age: "<<age<<endl;
```

```
    cout<<"          sex: "<<sex<<endl<<endl;
  }

  int main()
  { char c;
    Student s1;
    do{ cout<<endl<<"Please choice:"<<endl<<"  1---new_student"<<endl;
        cout<<"  2---display_student"<<endl<<"input: ";
        cin>>c;
        if (c=='1')        s1.new_s();
        else if (c=='2')   s1.view_s();
        cout<<" countiue ? (y/n) : ";
        cin>>c; cout<<endl;
      }while(c=='y'||c=='Y');
    return 1;
  }
```

11.5 指向类成员的指针

指向类成员的指针：不是指向类的对象、而是指向类的某个对象的某个成员的指针。
作用：通过指针访问对象的某个成员，包括数据成员和成员函数。
注意：除静态成员外，必须通过对象来确定类成员指针所访问的成员。

操作符 .* 和 ->* ：
通过对象或“引用”来确定*类成员指针*所访问的成员：.*
通过对象的指针来确定*类成员指针*所访问的成员：->*

11.5.1 指向类的数据成员的指针

1. 定义格式

格式：*被指向的成员的类型标识符 类名 :: *指针标识符;*
例：int A:: *pi;
char A:: *pc;

2. 例

```
#include<iostream.h>
#include<string.h>
class A{ public:
           int x,y,z;
           char c,s[8];
           A(int i=0,char cc='?',char *ss="name"):x(i),y(i),z(i),c(cc)
            { strcpy(s,ss); cout<<"new..."<<endl; }
           void show()
            { cout<<x<<" "<<y<<" "<<z<<" "<<c<<" "<<s<<endl; }
       };
void main()
{ int A::*pi;  //定义指针 pi 是指向类 A 的整型数据成员的指针
  char A::*pc; //定义指针 pc 是指向类 A 的字符型数据成员的指针
  pi=&A::x;    //设置指针 pi 指向 A 类的成员 x
  pc=&A::c;    //设置指针 pc 指向 A 类的成员 c
  A a1,a2(111,'w',"ok"),*pA=&a1;  //创建 A 类型对象 a1 和 a2，指针 pA
  a1.show();  a2.show();  cout<<endl;
   //利用操作符：.* ，通过指针 pi 和 pc 改变对象 a1 的值：
  a1.*pi=777;  a1.*pc='$';
```

```
    pi=&A::y;     a1.*pi=888; //改变指针 pi 的指向，指向成员 y，然后改变其值
    pi=&A::z;     a1.*pi=999; //改变指针 pi 的指向，指向成员 z，然后改变其值
            //通过指针 pi 和 pc 改变对象 a2 的值：
    a2.*pi=789;
    pi=&A::x;     a2.*pi=123;
    pi=&A::y;     a2.*pi=456;
    pc=&A::c;     a2.*pc='*';
    a1.show();  a2.show();  cout<<endl;
     //利用操作符：->* ，通过指针 pi 和 pc 改变对象 a1 的值：
    pi=&A::x;     pA->*pi=1;
    pi=&A::y;     pA->*pi=2;
    pi=&A::z;     pA->*pi=3;
    pc=&A::c;     pA->*pc='+';
          //通过指针 pi 和 pc 改变对象 a2 的值：
    pA=&a2; //pA 指向对象 a2
    pi=&A::x;     pA->*pi=4;
    pi=&A::y;     pA->*pi=5;
    pi=&A::z;     pA->*pi=6;
    pc=&A::c;     pA->*pc='/';
    a1.show();  a2.show();  cout<<endl;
}/*  执行结果：new ...
                new ...
                0 0 0 ? name
                111 111 111 w ok

                777 888 999 $ name
                123 456 789 * name

                1 2 3 + name
                4 5 6 / name     */
```

11.5.2 指向类的成员函数的指针

1. 定义格式

格式：*被指向的成员函数的返回类型（类名 :: *指针标识符）(形参列表)*;

例：void (A::*pfun)(int,double);

//定义指针 pfun 是指向 A 类的成员函数的指针，被指向的成员函数的返回类型为 void，有两个形参，类型分别为 int 和 double。

2. 例

```
#include<iostream.h>
#include<string.h>
class A{ public:
          int x,y,z;
            char c,s[8];
          A(int i=0,char cc='?',char *ss="name"):x(i),y(i),z(i),c(cc)
           { strcpy(s,ss); cout<<"new..."<<endl; }
          void show()
           { cout<<x<<" "<<y<<" "<<z<<" "<<c<<" "<<s<<endl; }
        };
void main()
{ void (A::*pfun)();
  pfun=A::show;
```

```
    A a1,a2(111,'w',"ok"),*pA=&a2;
    (a1.*pfun)();      //相当于：a1.show()
    (pA->*pfun)();
}/*  执行结果：new ...
              new ...
              0 0 0 ? name
              111 111 111 w ok     */
```

11.5.3 指向类的静态成员的指针

静态成员：无 this 指针，不必通过对象确定。

例题：

```
#include<iostream.h>
#include<string.h>
class A{ public:
       static int num;
       };
int A::num;  //静态数据成员，初始值为 0
void main()
{ int *p=&A::num;
  *p=56;                  //静态成员 num 的值为 56
  cout<<A::num<<endl;     //输出：56
  cout<<*p<<endl;         //输出：56
  cout<<p<<endl;          //输出：静态成员 num 的地址值
  A a,b;
  cout<<a.num<<endl;      //输出：56
  cout<<b.num<<endl;      //输出：56
}
```

11.6 保护成员

面向对象：在类中设置保护屏障，外界不能访问内部的成员，只能通过接口访问内部数据成员、或被保护的成员函数。

private,protected 段中的成员均是被保护的

private,protected 的区别在类的继承机制中被体现

11.7 屏蔽类的内部实现

即：用户无需了解内部的操作，只要知道“接口”即可。

用户只要了解类的公共成员

p /251 的最后 2 段

11.8 再论程序结构 p/252

1. 类的作用域

类的作用域：类的定义（包括成员函数的）的有效范围

一个类的所有成员的作用域均在该类的作用域内

一个类的成员函数在类作用域内，对该类的成员具有不加限制的访问权

对于本类作用域外的其他成员的访问，受“程序员”的控制（即：取决于编程）

p /252 的例题

2. 可见性

可见即可用

由于允许类型名与非本类的对象同名，所以要访问被屏蔽的对象时，可以：　（p /253）

1. 如果非类型名屏蔽了类型名，在类型名前冠以“class”即可使用被屏蔽了的类型名
2. 如果类型名屏蔽了非类型名，在非类型名前冠以作用域区分符“::”即可使用被屏蔽了的对象

局部类：在每个函数内定义的类

作用域：所属函数内

注意：①局部类的成员函数必须定义成内联函数，否则违反在函数中不得定义其他函数的规则；如果把成员函数的定义放在该函数外，则这个局部类无法与其联系

②局部类不常用

名空间：每个名字必须具有唯一的作用域

C++ 规定：①类型不能同名

②同一作用域的对象不能同名

③类、非类不属于同一名空间，所以类型名可以与其他非类型的对象同名，利用关键字 class 和作用域区分符（::）可以予以区别

3. 类的封装

将数据与使用这些数据的函数封装在类中，即：数据与操作结合构成不可分割的整体（类的对象）

数据和操作可以被保护，使得外界不可访问，只能通过接口访问

4. 类库的构成

C++ 类库包括类定义、成员函数定义。类定义以头文件方式提供给用户，成员函数定义以编译实现的代码方式提供。

5. C++ 程序结构

程序抽象：分析任务、区分不同的功能、抽象出各程序单元－－－模块

模块：从编译角度，模块是可以独立编译的程序单元

模块 { 规范说明、定义部分：.h　（描述与其他模块的接口）
实现部分：.cpp（模块的实现）

模块：实现信息隐藏的手段（访问控制：限制类的成员的可访问性；作用域规则：对象的有效域）

模块：抽象数据类型的实现工具，也是实现程序模块化的工具

对 p /256 的举例进行说明

12　构造函数与析构函数

类的 2 个特殊成员函数：构造函数、析构函数

12.1　类与对象

类：抽象的

对象：具体的，是类的一个具体的实例。对象有属性和操作

对象的定义：在 11.4 节中已介绍

对象的初始化：构造、并初始化对象

12.2　构造函数和析构函数

由于类的封装性，类的对象的初始化的任务只能由类的成员函数完成：构造函数

构造函数：在类定义体中，与类同名的成员函数
作用：在创建对象时，由系统自动调用（在编译时，由编译器自动完成），对对象进行初始化

析构函数：在类定义体中，与类同名、并在前面冠以“～”的成员函数
作用：释放对象在构造中分配的资源（例如：关闭被构造函数打开的文件，释放动态内存）。在对象生命期结束时，由系统自动调用，并且以与构造函数相反的调用顺序被调用。

例：//配合 12.2 节 (构造函数和析构函数)的讲解例题

```
#include<iostream.h>
#include<string.h>
int i=1;
class Abc{  char a[10];
            int*p1,*p2;
          public:
            Abc()
              { cout<<"constructor...i="<<i<<endl;
                strcpy(a,"ShangHai");
                p1=new int; *p1=100+i;
                p2=new int; *p2=200+i;
                ++i;
              }
            ~Abc()
              { cout<<"destructor... ";
                cout<<"*p1="<<*p1<<"   *p2="<<*p2<<endl;
                delete p1; delete p2;
              }
            void print()
              { cout<<"a="<<a<<"   *p1="<<*p1<<"   *p2="<<*p2<<endl;
              }
        };

int main()
{ Abc a1,a2; //执行构造函数
  a1.print(); a2.print();
  return 0;  //执行析构函数
}
/*执行结果：constructor...i=1
            constructor...i=2
            a=ShangHai   *p1=101   *p2=201
            a=ShangHai   *p1=102   *p2=202
            destructor... *p1=102   *p2=202  //自动调用，析构函数与构造函数的调用次序相反
            destructor... *p1=101   *p2=201  */
```

小结：
构造函数、析构函数是 2 个特殊的成员函数，应严格遵守其功能特性，不要在其中执行其他操作。不要调用它们。

构造函数与析构函数的共同点：

1.都是特殊的公有成员函数，与类同名；不过析构函数前面加：～

2.没有函数返回值，没有函数返回类型说明
但是，在函数内部可以使用 return 语句，从函数内提前返回（实际上隐含有返回值，此值供系统内部使用）

3.由系统自动调用：对象创建时调用构造函数，对象生存期结束前调用析构函数

4.不能被继承
5.不可取址（即：不可取构造函数和析构函数的地址）
6.如果不定义构造函数、析构函数，C++ 编译系统为每个类自动提供构造函数和析构函数：
如果不定义构造函数：系统自动生成一个不带参数的、什么也不做的公有构造函数：类名::类名(){ }
如果不定义析构函数：系统自动生成一个什么也不做的公有析构函数：类名::~类名(){ }
7.一旦定义了构造函数、析构函数，C++ 编译系统就不再为每个类提供构造函数和析构函数

构造函数与析构函数的不同点：

1.功能不同：构造函数创建对象，析构函数释放对象创建时所占的资源
2.构造函数可以带参数，析构函数不可以带参数
3.构造函数可以重载，析构函数不可以重载
4.构造函数不可以是虚函数，析构函数可以是虚函数
5.有拷贝构造函数，无拷贝析构函数
6.析构函数是执行构造函数的逆操作

12.3 带参数的构造函数

不带参数的构造函数对数据成员的初始化，取决于构造函数体中对哪些数据成员进行赋初值，且用此构造函数构造的所有对象的初始值都相同。

带参数的构造函数可以使得各对象的数据成员的初始值不同，通过创建对象时将实参传递给构造函数，达到初始化目的。

例：//配合 12.3 节 (带参数的构造函数)的讲解例题

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

class Abc{  char a[10];
            int*p1,*p2;
          public:
            Abc(char aa[],int *pp1,int *pp2)
             { strcpy(a,aa);
               p1=pp1;
               p2=pp2;
               cout<<"constructor..."<<a<<" "<<*p1<<" "<<*p2<<endl;
             }
            void print()
             { cout<<"a="<<a<<"   *p1="<<*p1<<"   *p2="<<*p2<<endl;
             }
          };

int main()
{ int x1=777,y1=999,x2=222,y2=888;
  Abc a1("ShangHai",&x1,&y1),a2("JiaoTong",&x2,&y2); //执行构造函数
  a1.print(); a2.print();
  return 0;  //执行析构函数
}
/*执行结果：constructor...ShangHai 777 999
            constructor...JiaoTong 222 888
            a=ShangHai   *p1=777   *p2=999
            a=ShangHai   *p1=222   *p2=888 */
```

12.4 重载构造函数

即：在一个类中出现多个构造函数，它们所带的参数个数、参数类型有所不同

例：如果在上例的 mian() 函数中创建对象 a3：

```
Abc a1("ShangHai",&x1,&y1),a2("JiaoTong",&x2,&y2),a3;
```

就必须重载构造函数：Abc()

```
{ strcpy(a,"abcdefg"); }
```

12.5 默认构造函数

即：系统自动提供的无参构造函数，该函数只创建对象，其他什么也不做。该函数创建的对象的初始值由对象的存储类型决定（全局的、静态的、局部的）

12.6 类成员初始化的困惑

当一个类中的某个成员是另一个类的对象时，如何初始化这个成员？

p /278 例题讲解

例：p/280 的例题

```
#include "iostream.h"
#include "string.h"
class StudentID{public:
                  StudentID(int id=0)
                    { value=id;
                      cout<<"Assigning student id:"<<value<<endl; }
                  ~StudentID()
                    { cout<<"Destrucing id:"<<value<<endl;  }
                protected:
                  int value;
                };
class Student{public:
                 Student(char *pname="noname",int ssID=0)
                  { cout<<"constructing student:"<<pname<<endl;
                    strncpy(name,pname,sizeof(name));
                    name[sizeof(name)-1]='\0';
                    StudentID id(ssID);  //实际上创建了一个局部对象 id
                  }
               protected:
                 char name[20];
                 StudentID id;
              };
void main()
{ Student s("Randy",9818); }
/* 执行结果：Assigning student id:0   //调用 StudentID 构造函数创建类成员：id，但用的缺省值：0
          Constructing student:Randy
          Assigning student id:9818
          Destructing id:9818
          Destructing id:0            */
```

12.7 构造类成员

即：初始化类成员

1. 一般变量的初始化

例：int m=10; 或者：int m(10); 此方式在新版 C++ 中允许

注意：赋值表达式中不允许：m(10); 只能：m=10;

2. 类的成员的初始化

把 12.6 节的例题中的 Student 类的构造函数改成如下，执行结果就是所要求的：

```
Student(char *pname="noname",int ssID=0):id(ssID)
```

```
        { cout<<"constructing student:"<<pname<<endl;
          strncpy(name,pname,sizeof(name));
          name[sizeof(name)-1]='\0';
          //StudentID id(ssID);
        }
```

执行结果：Assigning student id:9818
Constructing student:Randy
Destructing id:9818

类成员的初始化完全可以利用构造函数完成，在构造函数的原型中利用冒号语法，在冒号后面进行成员初始化：

构造函数（参数列表）：成员初始化列表
{.....}

例：

```
#include "iostream.h"
#include "string.h"
class A{   int x,y;
         public:
               A(int xx=0,int yy=0):x(xx),y(yy)
                 { cout<<"A... x="<<x<<" y="<<y<<endl; }
               void view()
                 { cout<<"x="<<x<<" y="<<y; }
       };
class B{   int x;
           A a;
        public:
           B(int xx=0,int ax=0,int ay=0):x(xx),a(ax,ay)
            { cout<<"B... x="<<x<<" A:"; a.view(); cout<<endl; }
       };
void main()
{ A a1,a2(12),a3(1,3); //或者：A a1,a2=12,a3(1,3);
  B b1,b2(789,111,333);
}  /* 执行结果：A... x=0 y=0
                A... x=12 y=0
               A... x=1 y=3
               A... x=0 y=0
               B... x=0 A:x=0 y=0
               A... x=111 y=333
               B... x=789 A:x=111 y=333  */
```

在B类中不能直接访问其他类的私有成员，只能通过接口（A类的公有成员）进行访问

12.8 程序中构造对象的顺序

1. 局部、静态对象

按在程序中定义的次序进行构造（但必须要符合语法）。

例：p /258 例题：

```
#include "iostream.h"
void main()
{ int m=5,n;
  if(m==5)  goto abc;
  n=m+1;
 abc:
  cout<<"m="<<m<<", n="<<n<<endl;
}/* 执行结果：m=5, n=-858993460     n是一个不确定的数 */
```

说明：对于静态对象只被构造一次。 p /286 例题

2. 全局对象

所有的全局对象都在主函数 main() 运行前被构造。在单个文件的程序中，全局对象按定义的顺序进行构造。

3. 类的成员

按成员在类中定义的顺序进行构造，而不是按构造函数中的初始化列表中的顺序进行构造。
例：p/287

13 面向对象程序设计 p/290

面向对象：抽象、分类
面向对象程序设计的目的：使用户既不需要懂得太多的计算机、也不需要懂太多的业务。

13.1 抽象

抽象与具体相对应
抽象是描述具体实物的必要手段

13.2 分类

层层分类－－－细化概念－－－具体化
归类：逐步抽象的过程

13.3 设计和效率

时、空
面向对象：高效、可读性、可维护性好

13.4 讨论 Josephus 问题

p/293

13.5 Josephus 问题的结构化方法的实现

按功能划分 p/294～298

13.6 Josephus 问题的面向对象方法的实现

按对象分割、抽象、归类
p/298～306 ：例题讲解

面向对象编程：用户不必了解内部实现的细节，只要了解解决 Josephus 问题的“接口”即可。
面向对象程序设计，编程人员分两类：应用程序设计及编程，类库设计及编程

13.7 Josephus 问题的程序维护

p/306：应用程序的功能有所扩展，即：程序维护
面向对象：不必改动原有程序，增加类的成员函数的定义；用户更不必关心程序如何改动、

14 堆与拷贝构造函数 p/312

14.1 关于堆

堆区：动态分配

操作符：new 动态分配内存
delete 释放由 new 分配的动态内存

堆内碎块：堆区内众多不连续的内存小块

动态分配内存应注意：及时释放内存，避免堆区碎块

14.2 需要 new 和 delete 的原因

C++ ：对象建立时应通过构造函数分配空间、构造数据结构、初始化数据。
而用 malloc()函数分配空间时，不自动调用构造函数；free()函数也不自动调用析构函数。
所以，需要增加操作符：new 和 delete，完成 C++规定的操作

14.3 分配堆对象

堆对象的作用域：文件
用 new 分配的对象，在用 delete 释放前，一直占有堆空间，不管能否访问得到。

堆对象创建时（new）自动调用构造函数，释放时（delete）自动调用析构函数

分配堆对象：
形式 1：指针标识符 = new 类型标识符；
形式 2：指针标识符 = new 类型标识符(初始化值)；
形式 3：指针标识符 = new 类型标识符[数组维数]； //对于数组，不能初始化

释放堆对象：
形式 1：delete 指针标识符；
形式 2：delete[]指针标识符；//释放堆数组

例：

```
#include "iostream.h"
static int k;
class A{  int x,y;
        public:
          A(int xx=0,int yy=0):x(xx+k),y(yy+k)
           { k++; cout<<"A..."<<x<<" "<<y<<endl; }
          ~A()
           { cout<<"~A.."<<x<<" "<<y<<endl; }
          void put()
           { cout<<x<<" "<<y<<endl; }
       };
void main()
{ int *p1,*p2=new int(789),*p3=new int[10];
  p1=new int; *p1=123;
  cout<<*p1<<" "<<*p2<<endl;

  A *pa1=new A(555,777),*pa2=new A[2];
  cout<<"pa1->put() :     "; pa1->put();
  cout<<"pa2->put() :     "; pa2->put();
  cout<<"(pa2+1)->put() : "; (pa2+1)->put();
  delete pa1;
  delete[]pa2;
}/* 执行结果：123  789
              A... 555 777
              A... 1 1
              A... 2 2
              pa1->put() :     555  777
              pa2->put() :     1  1
```

```
(pa2+1)->put() : 2  2
~A.. 555 777
~A.. 2 2
~A.. 1 1                    */
```

说明：1.动态分配堆数组对象时，不能对数组进行初始化
2.动态分配堆数组对象时，每维的大小必须指定，不能缺省

14.4 拷贝构造函数

拷贝构造函数：特殊的构造函数
作用：当用一个已存在的对象构造一个新的对象时，系统不自动调用构造函数、而是自动调用拷贝构造函数；创建新的对象，并把对象的数据成员值拷贝给新的对象（在编译时进行）。

拷贝构造函数声明的一般形式：类名(const 类名&形参对象名);
拷贝构造函数定义的一般形式：类名::类名(const 类名&形参对象名)
{…
…
}

拷贝构造函数被调用的3种情况：
1.新建对象：class A{...};

```
void main()
{ A a;
  A a1(a);
}
```

2.实参传递给形参时：...

```
void f(A a)
{ ... }
void main()
{ A a;
  f(a);
}
```

3.函数返回时：...

```
A fn()
{ A a;
  ...
  return a;
}
void main()
{ f(fn()); }
```

14.5 缺省的拷贝构造函数

如果不定义拷贝构造函数，系统自动为类提供一个缺省的拷贝构造函数（逐一拷贝数据成员）。

一般情况下，不必定义拷贝构造函数。但是，如果构造函数中存在动态分配，则必须定义拷贝构造函数

例：

```
#include<iostream.h>
#include<stdio.h>
class A{ private: int *p;
         public:  A(int i){ p=new int(i); cout<<"new..."<<endl; }
                  ~A(){ cout<<"delete p..."<<endl; delete p; }
        };
int main()
```

```
{ A a1(5);  //调用构造函数
  A a2(a1); //调用系统提供的缺省的拷贝构造函数
  return 0; //调用析构函数两次，一个值为 5 的 int 型对象被删除两次，在运行时会产生问
题
}
/*执行结果：new...
            delete p...
            delete p...      执行到此，出错：指针 p 已经被释放      */

//定义拷贝构造函数来解决上述问题
#include<iostream.h>
#include<stdio.h>
class A{ private: int *p;
         public:
             A(int i){p=new int(i);cout<<"new..."<<endl;}
             A(const A& r){p=new int(*r.p);cout<<"copy_consructor..."<<endl;}
             ~A(){delete p;cout<<"delete p..."<<endl;};
       };
int main()
{ A a1(5);
  A a2(a1); //调用自定义的拷贝构造函数
  return 0;
}
/*执行结果：new...
            copy_constructor...
            delete p...
            delete p...      */
```

14.6 浅拷贝与深拷贝

浅拷贝：拷贝成员，不拷贝资源
深拷贝：拷贝成员，也拷贝资源

p /320～322 例题讲解

如果类对象需要用析构函数来析构资源，必须定义具有析构资源功能的拷贝构造函数

14.7 临时对象

如果一个被调用函数有返回值，系统将产生一个同类型的临时对象来存放该返回值，然后把临时对象拷贝给调用函数，拷贝完毕，临时对象消失。
所以，函数的返回值不能是：对临时对象的“引用”。 p /323
即：对于临时对象，在整个创建它的外部表达式范围内有限，否则无效。

14.8 无名对象

无名对象：实际存在、但没有标识符的对象

可以利用构造函数创建无名对象
无名对象的 3 种用法：初始化一个“引用”、初始化一个对象的定义、函数的实参
p /324

14.9 构造函数用于类型转换

利用构造函数，由系统自动进行类型转换。
例：#include<iostream.h>
#include<string.h>
class A{ private: char name[20];

```
        public:
          A(char n[])
           { strcpy(name,n); cout<<"A..."<<endl; }
          void view()
           { cout<<"name: "<<name<<endl; }
      };
void fn(A a)
    { a.view(); }
void main()
{ char a[20]="Zhang qiushui";
  fn(a);    //自动利用 A 类的构造函数把 a 转换成：A 类
}
/*执行结果：A...
            name: Zhang qiushui   */
```

利用构造函数自动进行类型转换应注意：

系统只尝试含有一个参数的构造函数

如果存在二义性，系统立即放弃进行类型转换的尝试工作

二义性：不能唯一确定被调用的构造函数　p /325

15 静态成员与友元　　p/320

数据共享途径：函数参数传递

全局变量－－－具有危害性：破坏程序结构、面向对象的思想

静态成员

某个类的静态成员：该类的所有对象共同拥有并共享

声明格式：static 成员声明；

静态成员：静态数据成员、静态成员函数

15.1 静态成员的需要性

目的：类的所有成员共享

需要性：p /330

15.2 静态数据成员

静态数据成员：在类的定义体中，前面被冠以“static”的数据成员

说明：1.如果一个类中说明了静态数据成员，只有在这个类的第一个对象被创建时被初始化，自第二个对象起均不作初始化

2.静态数据成员是该类所有对象所共有的一个数据成员，其值可以被任何一个对象所改变

3.静态数据成员必须在使用前进行初始化，且初始化必须在类的定义体外进行：

格式：　数据类型 类名::静态数据成员名＝初值；

例：

```
#include<iostream.h>
class A{   int a;
           static int x;
         public:
           A(int aa=0):a(aa)
              { x++; cout<<"A...x="<<x<<endl; }
           static void view()
              { cout<<"x="<<x<<endl; }
       };
int A::x(900);   //或者：int A::x=900;
```

```
void main()
{ A a1,a2;
  a1.view(); a2.view();
}
/*执行结果：A...901
          A...902
          x=902
          x=902   */
```

15.3 静态成员函数

静态成员函数：在类的定义体中，前面被冠以“static”的成函数员

说明：1.静态成员函数不与任何对象联系，不存在 this 指针

2.一般的成员函数通过 this 指针进行调用，静态成员函数不能通过 this 指针进行调用。静态成员函数一般通过类名进行调用；也可以用对象名限定、进行调用，但是其作用仅是：通过对象名间接地指出类名

3.在静态成员函数中，基本上都是访问静态数据成员或全局变量，不能对类的非静态成员进行默认访问。如果要访问非静态数据成员，利用参数传递进行（以限定哪个对象的数据）

15.4 静态成员的使用

例：静态成员函数、静态数组及其初始化

```
#include<iostream.h>
#include<stdio.h>
class A{   static int a[20];
           int x;
         public:
           A(int xx=0){ x=xx; }
           static void in();
           static void sh();
           void show(){ cout<<"x="<<x<<endl; }
        };
int A::a[20]={0,0};
void A::in()
  { cout<<"input a[20]:"<<endl;
    for(int i=0;i<20;++i)
        cin>>a[i];
  }
void A::sh()
  { for(int i=0;i<20;++i)
     cout<<"a["<<i<<"]="<<a[i]<<endl;
  }
void main()
{ A::in();  //调用静态成员函数 in()，初始化静态数组 a[20]
  A::sh();  //调用静态成员函数 sh()，显示静态数组 a[20]
  A a;      //定义 A 类的对象 a
  a.sh();   //与 A::sh()同
  a.show(); //显示对象 a 的数据成员值
}
```

p /331 例题讲解

15.5 需要友元的原因 p/341

问题的提出：

对象的私有成员：只允许本类的成员函数访问

↓

希望：本类以外的对象或函数能够访问类中的私有成员

↓

友元：提供了本类外的对象访问私有成员的途径

友元：一个类的友元可以访问这个类的私有成员（公有成员当然能够访问）。
某个类的友元：肯定不属于这个类的成员。
友元可以是下列之一：友元函数－－－不属于任何类的一般函数
友元成员－－－另一个类的某个成员函数
友元类　－－－另一个类（整个类作友元）

注意：友元使得数据的封装性受到影响，程序的可维护性变差，应慎重使用

15.6 友元的使用

1. 友元的说明和定义

友元声明：一个类的友元，必须在该类的定义体中的公有段予以声明，并在函数声明前冠以“friend”
友元的定义：友元不属于本类的成员，所以友元的定义不能在本类的定义体中。

2. 友元函数

友元函数：不属于任何类的、是某个类的友元的一般函数
说明：由于友元函数不是该类的成员函数，没有 this 指针，不能确定当前访问的是哪个对象的成员，所以，友元函数应该带有传递对象的参数，以确定访问哪个对象的数据

例：

```
#include<iostream.h>
#include<stdio.h>
class A{  int x,y;
        public: int z;
          A(int xx=0,int yy=0,int zz=0):x(xx),y(yy),z(zz){}
          friend void in(A);
          friend void sh(A);
        };
void in(A a1)
  { cout<<"input x,y: ";
    cin>>a1.x>>a1.y>>a1.z;
    cout<<a1.x<<" "<<a1.y<<" "<<a1.z<<endl;
  }
void sh(A a1)
  { cout<<a1.x<<" "<<a1.y<<" "<<a1.z<<endl; }
void main()
{ A a1(11,22,33);  sh(a1);   in(a1);   sh(a1); }
/* 执行结果：11 22 33
             input x,y:   //输入：777 888 999
             777 888 999
             11 22 33     //不改变类 A 的对象 a1 的值   */
```

例：p/344 例题

3. 友元成员

友元成员：是其他某个类的成员函数，是本类的友元
格式：friend　函数返回类型　类名标识符::函数名(参数列表)；

说明：1.友元成员的声明，除了前面冠以“friend”外，还要注明所属类的类名
2.友元成员应该在自己所属类的定义体中进行成员函数声明（或定义）

3.友元成员的 this 指针是指明自己类的对象的调用;要访问把自己声明为友元的那个类的私有成员，应该带有传递对象的参数，以确定访问哪个对象的数据

例：p/345 例题

4. 友元类

友元类：是一个类，而且是另一个类的友元
友元类的说明：friend class 类名;

说明：类 A 和 B，B 被声明为 A 的友元类，则 B 类的所有成员函数都可以访问 A 类的私有成员

```
例：#include<stdio.h>
#include<string.h>
class A{  int x,y;
        public:
              A(int xx=0,int yy=0):x(xx),y(yy){}
          friend class B;
        };
class B{  char n[20];
        public:
              B(char nn[]){ strcpy(n,nn); }
              void sh(A a1)
                { cout<<n<<": "<<a1.x<<" "<<a1.y<<endl; }
              void sum(A a1)
                { cout<<n<<": "<<a1.x+a1.y<<endl; }
           };
void main()
{ A a1(111,222);
  B b1("abcdefg");
  b1.sh(a1);
  b1.sum(a1);
}/* 执行结果：abcdefg: 111 222
                  abcdefg: 333     */
```

例：p/346 例题

16 继承

继承：为一个新的类提供已有类的数据结构和操作
继承是 C++ 的 3 大重要特性之一
作用：达到可重用性、可扩充性的目的，避免程序的重复设计、重复开发。

16.1 继承的概念

基类（父类）： 派生新类的类
派生类（子类）：从基类派生而成的类
基类和派生类：构成类的层次关系

单继承：从一个基类派生而成的类
多继承：从多个基类派生而成的类

继承类别：公有继承、私有继承、保护继承
访问控制：决定子类对父类的访问权限

继承类别与访问控制：

访问控制	继承类别	
public	公有继承	
private	私有继承	（17.5节 17.6节：p/389 p/391）
protected	保护继承	（17.5节 17.6节：p/389 p/391）

关于继承的说明：

1.如果子类继承了父类，则子类自动具有父类的全部数据成员（数据结构）和成员函数（功能）；但是，
子类对父类的成员的访问有所限制：
父类的私有成员只能被父类自己的成员函数、友元访问，任何子类均不能直接访问；
子类对父类的公有成员的访问权限取决于继承时的“访问控制”的设定

2.子类可以定义自己的成员：数据成员和成员函数

3.基类、派生类，父类、子类都是“相对”的

16.2 继承的工作方式

1. 派生类的定义

语法：class 派生类名:访问控制 基类名{ …
…
…
};

访问控制：为private、public、protected之一，缺省为：private

2. 继承方式

⑴公有继承
基类的公有段成员被继承为公有的
基类的保护段成员被继承为保护的

派生时用“public”作访问控制

⑵私有继承 （17.6节： p/391）
基类的公有段成员被继承为私有的
基类的保护段成员被继承为私有的

派生时用“private”作访问控制

⑶保护继承 （17.6节： p/391）
基类的公有段成员被继承为保护的
基类的保护段成员被继承为保护的

派生时用“protected”作访问控制

p /350 例题讲解

16.3 派生类的构造

例：//配合16.3节(继承 p/352: 派生类的构造)的讲解例题

```
#include<iostream.h>
#include<stdio.h>
#include<string.h>
class people                    //人类
     {  long number;            //身份证号(假设long型，其长度够了)
        char name[20],sex; //姓名，性别
      public:
        people(long num=0,char* n="\0",char s='m')
```

```
        { number=num; strcpy(name,n),sex=s; }  //构造函数
      void p_show()
        { cout<<"people: number="<<number;
          cout<<"  name="<<name<<"  sex="<<sex<<endl; }
   };
class student:public people  //学生类，具有基类 people 类的全部数据结构和操作
   {  int s_num;            //学号
    public:
      int s_class;          //班级
      student(long n,char* na,char s='m',int sn=0,int sc=0):people(n,na,s)
        { s_num=sn; s_class=sc; }          //构造函数
      void s_show()
        { cout<<"student: "<<endl;;
          p_show();
          cout<<"        s_num="<<s_num<<"     s_class="<<s_class<<endl; }
   };
class member:public people //教工类，具有基类 people 类的全部数据结构和操作
   {  int m_num;         //工号---私有成员
    public:
      char department[10];  //部门---公有成员
      member(long n,char* na,char s='m',int mn=0,char* md="\0"):people(n,na,s)
        { m_num=mn; strcpy(department,md); }   //构造函数
      void m_show()
        { cout<<"member:"<<endl;
          p_show();
          cout<<"        m_num="<<m_num<<"     department="<<department<<endl;}
   };
class worker:public member //工人类，具有基类 people 和 member 的全部数据结构和操作
   {  char station[10];  //岗位
    public:
      worker(long n,char* na,char s='m',int mn=0,char* md="\0",char* st="\0"):\
            member(n,na,s,mn,md)
        { strcpy(station,st); }   //构造函数
      void w_show()
        { cout<<"worker:"<<endl;
          m_show();
          cout<<"        station="<<station<<endl; }
   };
class teacher:public member//教师类，具有基类 people 和 member 的全部数据结构和操作
   {  char course[10];   //执教课程
    public:
      teacher(long n,char* na,char s='m',int mn=0,char* md="\0",char* tc="\0"):\
            member(n,na,s,mn,md)
        { strcpy(course,tc); }    //构造函数
      void t_show()
        { cout<<"teacher:"<<endl;
          m_show();
          cout<<"        course="<<course<<endl; }
   };

void main()
{ people p(981102,"p_name",'w');
  p.p_show();
  student s(781010,"s_name",'m',1001,982);
```

```
    s.s_show();
    worker w(123456,"w_name",'m',1111,"factory","smith");
    w.w_show();
    teacher t(661001,"t_name",'m',1954,"computer","c++");
    t.t_show();
    t.m_show();                 //公有继承的派生类对象直接访问基类的公有成员
    t.p_show();                 //公有继承的派生类对象直接访问基类的基类的公有成员
    cout<<t.department<<endl;  //直接访问基类的公有数据成员
} /*执行结果：people:  number=981102  name=p_name  sex=w
              student:
              people:  number=781010  name=s_name  sex=m
                       s_num=1001     s_class=982
              worker:
              member:
              people:  number=123456  name=w_name  sex=m
                       m_num=1111     department=factory
                       station=smith
              teacher:
              member:
              people:  number=661001  name=t_name  sex=m
                       m_num=1954     department=computer
                       course=c++
              member:
              people:  number=661001  name=t_name  sex=m
                       m_num=1954     department=computer
              people:  number=661001  name=t_name  sex=m
              computer   */
```

继承中的构造函数和析构函数的说明：

1. 派生类的构造函数的初始化列表中列出的均是直接基类的构造函数。
2. 构造函数不能被继承，因此派生类的构造函数只能通过调用基类的某个构造函数（如果有重载的话）来初始化基类子对象。
3. 派生类的构造函数只负责初始化自己定义的数据成员。
4. 先调用基类的构造函数，再调用派生类自己的数据成员所属类的构造函数，最后调用派生类的构造函数；派生类的数据成员的构造函数被调用的顺序取决于在类中声明的顺序。
5. 析构函数不可以继承，不可以被重载。
6. 派生类的对象的生存期结束时调用派生类的析构函数，在该析构函数结束之前再调用基类的析构函数；所以，析构函数的被调用次序与构造函数相反。

16.4 继承与组合

组合（也称“聚集”）：新的、复杂的对象由已存在的、相对简单的对象组成
即：类的数据成员是另一个类的对象

对象：由若干个子对象组成
聚集：“工程”的体现，预制件的组合
关键问题：在建立这种类型的对象时，如何对子对象进行初始化

利用构造函数进行初始化：

类名::类名(参数列表):子对象 1(初始值列表),子对象 2(初始值列表),...,子对象 n(初始值列表)

```
{ ...
  ...
};
```

16.5 多态性

多态性：面向对象系统中的又一重要概念，3 大特性之一。多态性是通过函数重载来实现的。
多态性：当不同的对象收到相同的消息时产生不同的动作（本书的多态性概念：运行时确定调用哪个函数的能力称为多态性）

面向对象系统的二种编译方式：
早期联编－－－系统在编译时就确定如何实现某一动作，提供了执行速度快的优点
滞后联编－－－系统在运行时动态实现某一动作，提供了灵活和高度抽象的优点
早期联编和滞后联编都支持多态性

C++ 系统支持二种多态性：静态多态性－－－编译时的多态性
动态多态性－－－运行时的多态性

1. 静态多态性

通过函数重载实现
函数重载的二种方式：在同一个类中重载
基类成员在派生类中的重载
⑴在同一类中重载：同名的成员函数，根据参数等不同，自动予以区别
⑵基类成员在派生类中的重载：

```
class A{ ...
        public:
          void show(){}
          ...
        };
class B:publi A
       { ...
        public:
          void show(){}
          void show(int xx){}
          ...
        };
void main()
{ A a1;
  B b1;
  a1.show();      //调用 A 类的 show()函数，操作数据是对象 a1 的
  b1.show();      //调用 B 类的 show()函数，操作数据是对象 b1 的
  b1.show(555);   //调用 B 类的 show(int xx)函数，操作数据是对象 b1 的
  b1.A::show();   //调用 A 类的 show()函数，操作数据是对象 b1 的
}
```

2. 动态多态性

C++ 中 ，运行时的多态性是通过虚函数重载来实现的

16.6 多态的思考方式

对于同一件事，对象不同，实际操作有所不同。
例如：学生注册收费，本科生、研究生所收的费用不同，注册盖章不同

p/357 例题

16.7 多态性如何工作

运行时的多态性通过虚函数实现
虚函数：是一种动态的函数重载的方式，虚函数允许函数调用与函数体之间的联系在运行时才建立。

1. 对象指针

可以通过指针访问对象

例：
```
class A{ ...
         public:
           show(){}
           ...
         }a1,*p=&a;
   ...
   p->show();  a1.show();
   ...
```

一般对象的指针：

各类的指针相互独立，不能混用

例：class A{...}a1,*pa;
class B{...}b1,*pb;

指针 pa 只能指向 A 类对象，不能指向 B 类对象；指针 pb 只能指向 B 类对象，不能指向 A 类对象

派生类的对象的指针：

派生类与基类相互之间既相互独立、又存在继承与被继承的关系，所以，派生类的指针与基类的指针之间相互有关。

2. 为什么引入虚函数

例：
```
#include"iostream.h"
class A{  int x,y;
        public:
          void show(){ cout<<"class A..."<<endl; }
          void view(){ cout<<"class A view()..."<<endl; }
       };
class B:public A
      {  int x,y;
       public:
         void show(){ cout<<"class B..."<<endl; }
         void sh(){ cout<<"class B sh()..."<<endl; }
      };
class C:public B
      {  int x;
       public:
         void show(){ cout<<"class C..."<<endl; }
         void sh(){ cout<<"class C sh()..."<<endl; }
      };
void main()
{ A a,*pa=&a;
  B b,*pb=&b;
  C c;
  //pb=&a;    //出错：派生类的指针不可以指向基类的对象
  pa->show();
  pa=&b;
  pa->show();
  //pa->sh(); //出错：基类指针可以指向派生类，但不可以通过该指针访问不属于基类的成员
  pa=&c;
  pa->show();
  ((B*)pa)->show();
  ((B*)pa)->sh();
```

```
((C*)pa)->show();
((C*)pa)->sh();
((C*)pa)->view();
} /* 执行结果：class A...
                class A...
                class A...
                class B...
                class B sh()...
                class C...
                class C sh()...
                class A view()...   */
```

执行结果分析：基类的指针可以指向派生类的对象，但是，通过指针对成员函数的调用，仅仅与指针本身的类型有关，与指针当前指向的对象无关。如果要与当前对象有关，必须对指针类型进行强制转换。

↓

希望：通过指针进行一种动态的指向

即：指针指向的对象不同，执行不同的重载成员函数

↓

引入虚函数：把上例中基类的 show()函数声明成虚函数即可（函数原型前加关键字：virtual）

注意：①一个声明为基类对象的指针可以指向它的公有派生类的对象，但是不能指向其私有派生类的对象

②一个声明为派生类的指针不可以指向其基类的对象

③一个声明为基类对象的指针，如果指向它的公有派生类的对象时，只能利用其来访问基类的被公有继承的成员，不能访问派生类自己的成员

④一个声明为基类对象的指针，如果指向它的公有派生类的对象时，可以通过对指针类型进行强制转换来访问派生类自己的成员

3. 虚函数的定义和使用

定义： 在函数原型前面加“virtual”即可

例 1：上面的例题，把类 A 的成员函数 show()改成虚函数，其他不作任何改动：

```
virtual void show(){ cout<<"class A..."<<endl; }
```

然后执行，执行结果如下：

```
class A...
class B...
class C...
class C...  //指针 pa 已经指向 c，而 show()是虚函数：((B*)pa)->show();
class B sh()...
class C...
class C sh()...
class A view()...
```

例 2：p /358 例题讲解

16.8 不恰当的虚函数

例 1：虚函数的原型必须相同（函数的返回类型不除外），才能实现多态性，否则不管是否冠以“virtual”声明，均作为一般重载函数处理，失去虚特性。虚特性可以隔代传递

```
#include<iostream.h>
#include<stdio.h>
class A{  int x,y;
        public:
```

```
            virtual void show(int a=0){cout<<"A: "<<endl; }   //声明为虚函数
          };
    class B:public A
          {public:
            virtual void show(){ cout<<"B: "<<endl; } //一般的成员函数重载，不是虚函数重
载
          };
    class C:public B
          {public:
            void show(int b=0){ cout<<"C: "<<endl; }   //虚函数重载
          };
    class D:public C
          {public:
            void show(int b=0){ cout<<"D:\n"; }        //虚函数重载
          };
    void main()
    { A a,*pa=&a;
      B b; C c; D d;
      pa->show();
      pa=&b; pa->show();  //执行基类 A 的 show()函数
      pa=&c; pa->show();
      pa=&d; pa->show();
    } /*执行结果：A:
                  A:
                  C:
                  D:  */
```

例 2：p/362 例题讲解

```
    #include<iostream.h>
    class Base{ public:
                  virtual Base* afn()
                    { cout<<"This is Base class.\n"; return this; }
              };
    class SubClass:public Base
        { public:
            Base* afn()
              { cout<<"This is SubClass.\n"; return this; }
        };
    void test(Base& x)
    { Base *b;
      b=x.afn();
    }

    void main()
    { Base bc;
      SubClass sc;
      test(bc);
      test(sc);
    } /*执行结果：This is Base class.
                  This is SubClass.     */
```

16.9 虚函数的限制

关于虚函数的说明：

1. 虚函数是引入继承机制后，用于表现基类和派生类之间的成员函数重载的一种关系，是特殊的函数重

载。
2. 关键字 virtual 只能对成员函数、且基类中的非静态的公有成员函数进行声明。
3. 虚函数的声明在基类中进行，虚函数具有传递性（可以隔代传递），派生类中可以省略关键字 virtual。
4. 基类中的某个成员函数被声明成虚函数后，此虚函数可以在其派生类中被重载，并可以被重新定义，但是函数原型必须相同（包括函数的返回类型），否则系统把其作为一般的函数重载而失去虚特性。
5. 内联函数不能声明成虚函数
6. 构造函数不能声明成虚函数
7. 析构函数可以声明成虚函数，且对具有继承关系的基类的析构函数应声明成虚函数
8. 在执行过程中，含有虚函数的基类的指针可以不断地改变指向，指向基类的派生类的对象，以执行当前指向的对象所属类的成员函数，从而实现动态多态性。

16.10 类的冗余

在一个软件（程序）中，如果类的划分不尽合理，便会造成信息或功能操作的冗余。因为，这些类毕尽是不同的类，如果各类中包含相同或相近的信息，这些信息以分离的形式出现－－－冗余。

p/364 例题讲解

16.11 克服冗余带来的问题

克服冗余：合理的继承

p/369 图 16-4

16.12 类的分解

分解：从不同的类中抽取并组合共有特征的过程。即把各类的共同特征抽取出来，然后把这些特征放到上面，作为这些类的基类。

目的：减少编程工作量，提高软件质量。

p/371 例题

16.13 抽象类

问题的提出：有时基类往往表示一种抽象的概念，这种类并没有具体的对象，其成员函数也并无具有实际意义的操作，这种类认为是一种抽象的类。

1. 纯虚函数

纯虚函数：基类中的永远不被调用执行的虚函数称为纯虚函数

纯虚函数的作用：利用其虚函数的传递特性，达到继承的动态多态性目的

纯虚函数的定义：virtual 函数返回类型 函数名(参数列表)=0;

说明：虚函数被声明成纯虚函数，就不必进行函数体定义

2. 抽象类

抽象类：至少包含一个纯虚函数的类称为抽象类

抽象类不能创建对象，是为众多派生类提供共同的数据结构和操作抽象定义

16.14 由抽象类派生具体类

p/377 例题讲解

16.15 纯虚函数的需要性

p/378 例题讲解

关于抽象类和纯虚函数的说明：
1. 抽象类只能作为其他类的基类，不能创建对象。
2. 抽象类不能用作函数形参的类型、函数返回类型，不能进行显式类型转换。
3. 可以定义抽象类的指针，此指针可以指向其非抽象类的派生类的对象，以实现动态多态性。

4.在抽象类的派生类中，如果没有给出基类的纯虚函数的具体定义，则该派生类仍是一个抽象类；如果在派生类的派生类中，还没有给出基类的纯虚函数的具体定义，则该派生类的派生类还是一个抽象类。

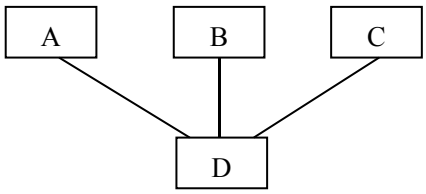
17 多重继承

多重继承即多继承：

由多个基类派生而成新的类

17.1 多继承如何工作

类A、B、C、D，继承结构图：



类D继承类A、B、C，类D是类A、B、C的派生类，类D包含类A、B、C的所有数据成员和数据结构

例：

```
#include<iostream.h>
class A{  int i;
        public:
          A(int ii=0){ i=ii; cout<<"A... i="<<i<<endl; }
          void show()
           { cout<<"A::show()  i="<<i<<endl;}
        };
class B{  int i;
        public:
          B(int ii=0){ i=ii; cout<<"B... i="<<i<<endl; }
          void show()
           { cout<<"B::show()  i="<<i<<endl; }
        };
class C:public A,public B
       {  int i;
        public:
          C(int i1=0,int i2=0,int i3=0):A(i1),B(i2)
           { i=i3; cout<<"C... i="<<i<<endl; }
          void show()
            {cout<<"C::show()  i="<<i<<endl;
             //show();//根据"支配规则"，调用自己的类的成员函数show()，出现死循环
            }
        };
void main()
{ C c(1,2,3);
  c.A::show(); //调用对象c的基类A的成员函数show()
  c.show();    //根据"支配规则"，调用对象c所属的类C的成员函数show()
} /*执行结果：A... i=1
              B... i=2
              C... i=3
              A::show()  i=1
              C::show()  i=3    */
```

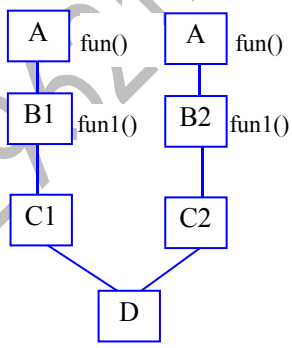
17.2 继承的模糊性

模糊性：二义性
二义性：对某个成员访问不能唯一确定

二义性产生的原因：不同类的成员可以同名

规定：在多继承、多层次的继承结构中，总是逐层往上、访问最靠近自己的那个同名成员。但是，如果在同一层的两个类中具有同名成员，则产生二义性。

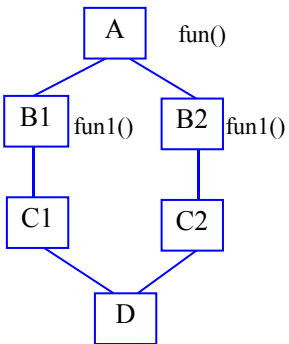
例：
```
#include<iostream.h>
class A{public:
          void fun(){cout<<"A:fun()"<<endl; }
       };
class B1:public A
       {public:
          void fun1(){cout<<"B1:fun1()"<<endl; }
       };
class B2:public A
       {public:
          void fun1(){cout<<"B2:fun1()"<<endl; }
       };
class C1:public B1
       {};
class C2:public B2
       {};
class D:public C1,public C2
       {};
void main()
{ D obj;
  //obj.fun1();       //不可以执行：二义性(B1 和 B2 是两个不同的类，各有 fun1()函数)
  obj.B1::fun1();
  obj.B2::fun1();
  //obj.fun();        //不可以执行：二义性(在 B1 和 B2 中都有 fun()函数的“副本”)
  //obj.A::fun();     //不可以执行：二义性
  obj.B1::fun();      //无二义性：可以执行
}/*执行结果：B1:fun1()
             B2:fun1()
             A:fun()     */
```



二义性解决办法：①在成员名前用类名进行限定（类名::成员名），可以解决同一层的两个类中具有同名成员而产生的二义性
②虚基类（即虚拟继承），可以解决上述访问同一个基类的成员、但存在多条路径的问题

17.3 虚拟继承

例：把上例的类 B1、B2 改成由 A 类虚拟继承而成
```
#include<iostream.h>
class A{public:
          void fun(){cout<<"A:fun()"<<endl; }
       };
class B1:virtual public A
       {public:
          void fun1(){cout<<"B1:fun1()"<<endl; }
```



```
        };
class B2:virtual public A
        {public:
           void fun1(){cout<<"B2:fun1()"<<endl; }
        };
class C1:public B1
        {};
class C2:public B2
        {};
class D:public C1,public C2
        {};
void main()
{ D obj;
  //obj.fun1();       //不可以执行：二义性(B1 和 B2 是两个不同的类)
  obj.B1::fun1();
  obj.B2::fun1();
  obj.fun();          //可以执行：无二义性
  obj.A::fun();       //可以执行：无二义性
  //obj.B1::fun();    //无二义性：可以执行
}/*执行结果：B1:fun1()
             B2:fun1()
             A:fun()
             A:fun()     */
```

虚拟继承的概念：在多条继承路径上存在一个公共的基类，在这些路径的汇合处（某个派生类）就产生这个公共基类的多个“副本”。如果只希望存在一个副本，可以把此公共基类声明为虚基类（虚拟继承）。

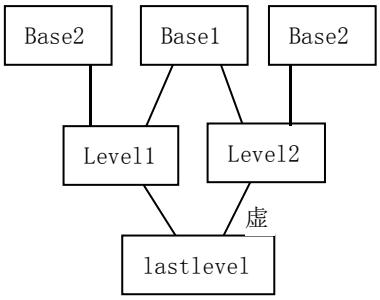
虚拟继承的方法：虚基类是对派生类而言，所以，虚基类本身的定义不变，在定义派生类时声明该基类为虚基类即可(冠以关键字：virtual)。

17.4 多继承的构造顺序

即：在多继承中创建派生类对象时，构造函数的调用顺序

例：

```
#include<iostream.h>
class base1{   int a,b;
             public:
               base1(int aa=0,int bb=0)  //构造函数：是带缺省值的，或者是无参数的
                { a=aa; b=bb;
                  cout<<"base1  class!"<<endl; }
               void show(){ cout<<"base1: a="<<a<<" b="<<b<<endl; }
            };
class base2{ public:
               base2(){ cout<<"base2  class!"<<endl; }
               void show2(){ cout<<"base2:  "<<endl; }
           };
class level1:public base2,virtual public base1
            { public:
                level1(int a,int b):base1(a,b)
                  { cout<<"level1 class!"<<endl; }
            };
class level2:public base2,virtual public base1
            { public:
                level2(int a,int b):base1(a,b)
```



```
                    { cout<<"level2 class!"<<endl; }
                };
    class lastlevel:public level1,virtual public level2
                { public:
                    lastlevel(int a,int b):level1(a,b),level2(a,b)
                      { cout<<"lastlevel class!"<<endl; }
                };
    void main()
    { lastlevel obj(7,9);
      obj.show();     //直接调用间接基类 base 的公有成员函数 show()---无二义性
      //obj.show2(); //访问间接基类 base2 的公有成员函数 show2()-----存在二义性
    }/*执行结果：base1  class!
                 base2  class!
                 level2 class!
                 base2  class!
                 level1 class!
                 lastlevel class!
                 base1: a=0 b=0     //见下面第 7 点    */
```

派生类的构造函数调用次序(调用原则)：

1. 先基类，再成员，后自己
2. 在同一层上如有多个基类，则先虚基类，后非虚基类
3. 在同一层上如有多个虚基类，则按派生时定义的先后次序执行
   在同一层上如有多个非虚基类，则按派生时定义的先后次序执行
4. 对于一个派生类的某个虚基类的构造函数一旦被执行过，就不再被多次执行
5. 虚基类的派生类的构造函数的定义无特殊规定
6. 如果“最派生类”的构造函数的初始化列表中不调用虚基类的构造函数，则该派生类的虚基类的构造函数必须是无参的,或全部是缺省值的构造函数。
7. 如果“最派生类”的构造函数的初始化列表中不调用虚基类的构造函数，则虚基类的初始值采用其构造函数的缺省值；如果“最派生类”的构造函数的初始化列表中调用了虚基类的构造函数，并且带入初始值，则虚基类的初始值采用带入的值。

17.5 继承类别及其访问控制

公有成员：一个类的公有成员允许本类的成员函数、本类的对象、公有派生类的成员函数、公有派生类的对象访问
私有成员：一个类的私有成员只允许本类的成员函数访问
保护成员：具有私有成员和公有成员的特性，对其派生类而言是公有成员，对其对象而言是私有成员

例 1：类的保护成员

```
#include<iostream.h>
class A{private:
          int i;
       protected:
          int j;
          void show1()
            { cout<<"show1(): i="<<i<<" j="<<j<<endl; }
       public:
          A(int x,int y){ i=x; j=y; }
          void show2()
            { cout<<"show2(): i="<<i<<" j="<<j<<endl; show1();} //成员函数可以访问保护成员
       };
int main()
```

```
{ A a1(7,9);;
  //a1.show1();  //不可以直接访问类的保护成员：show1()
  a1.show2();
  return 0;
}/* 执行结果：show2(): i=7 j=9
              show1(): i=7 j=9    */
```

公有继承：基类的公有段成员被继承为公有的，基类的保护段成员被继承为保护的。派生时用“public”作访问控制。

私有继承：基类的公有段成员被继承为私有的，基类的保护段成员被继承为私有的，派生时用“private”作访问控制。

保护继承：基类的公有段成员被继承为保护的，基类的保护段成员被继承为保护的，派生时用“protected”作访问控制。

公有继承的访问控制：对基类的公有段成员的访问，尤如访问自己的公有段成员；
对基类的保护段成员的访问，尤如访问自己的保护段成员。

私有继承的访问控制：对基类的公有段成员的访问，尤如访问自己的私有段成员；
对基类的保护段成员的访问，尤如访问自己的私有段成员。

保护继承的访问控制：对基类的公有段成员的访问，尤如访问自己的保护段成员；
对基类的保护段成员的访问，尤如访问自己的保护段成员。

例 2：类的私有继承

```
#include<iostream.h>
#include<string>
class people                  //人类
     {  long number;          //身份证号(假设 long 型，其长度够了)
        char name[20],sex; //姓名，性别
      public:
        people(long num=0,char* n="\0",char s='m')
          { number=num; strcpy(name,n);sex=s; }  //构造函数
        void p_show()
          { cout<<"people: number="<<number;
            cout<<"  name="<<name<<"  sex="<<sex<<endl; }
     };
class member:private people //教工类，具有基类 people 类的全部数据结构和操作
     {  int m_num;           //工号---私有成员
      public:
        char department[10];  //部门---公有成员
        member(long n,char* na,char s='m',int mn=0,char* md="\0"):people(n,na,s)
          { m_num=mn; strcpy(department,md); }   //构造函数
        void m_show()
          { cout<<"member:"<<endl;
            p_show();                          //访问基类的公有成员
            cout<<"        m_num="<<m_num<<"     department="<<department<<endl;}
     };
class worker:public member //工人类，具有基类 people 和 member 的全部数据结构和操作
     {  char station[10];  //岗位
      public:
        worker(long n,char* na,char s='m',int mn=0,char* md="\0",char* st="\0"):\
               member(n,na,s,mn,md)
          { strcpy(station,st); }   //构造函数
        void w_show()
          { cout<<"worker:"<<endl;
            m_show();               //访问基类的公有成员
```

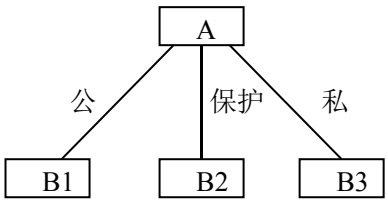
```
            cout<<"        station="<<station<<endl; }
    };
class teacher:private member//教师类，具有基类 people 和 member 的全部数据结构和操作
     {  char course[10];   //执教课程
      public:
        teacher(long n,char* na,char s='m',int mn=0,char* md="\0",char* tc="\0"):\
              member(n,na,s,mn,md)
          { strcpy(course,tc); }     //构造函数
        void t_show()
          { cout<<"teacher:"<<endl;
            m_show();                //访问基类的公有成员
            cout<<"        course="<<course<<endl; }
    };

void main()
{ worker w(123456,"w_name",'m',1111,"factory","smith");
  w.w_show();
  w.m_show();   //work 类公有继承 member，所以可以直接访问 member 类的公有成员
  //w.p_show(); //member 类私有继承 people，所以不可以直接访问 people 类的公有成员
  teacher t(661001,"t_name",'m',1954,"computer","c++");
  t.t_show();
  //t.m_show(); //teacher 类私有继承 member，所以不可以直接访问 member 类的公有成员
  //t.p_show(); //teacher 类私有继承 member，member 私有继承 people 类，不可以直接访问
} /*执行结果：worker:
              member:
              people:  number=123456  name=w_name  sex=m
                       m_num=1111     department=factory
                       station=smith
              member:
              people:  number=123456  name=w_name  sex=m
                       m_num=1111     department=factory
              teacher:
              member:
              people:  number=661001  name=t_name  sex=m
                       m_num=1954     department=computer
                       course=c++    */
```

例 3：类的保护继承

```
#include<iostream.h>
class A{private:
        int i;
      protected:
        int j;
        void show_A1()
          { cout<<"show_A1(): i="<<i<<" j="<<j<<endl; }
      public:
        A(int x,int y){ i=x; j=y; }
        void show_A2()
          { show_A1();
            cout<<"show_A2(): i="<<i<<" j="<<j<<endl; }
      };
class B:protected A
       {private:
          int x;
```

```
        public:
          B(int i,int j,int k):A(i,j)
            { x=k; }
          void show_B()
            { cout<<"show_B():  x="<<x<<endl; }
        };
class C:public B
       {public:
          C(int i,int j,int x):B(i,j,x) { }
          void show_C()
            { show_A2();
              show_B();
              cout<<"show_C():  "<<endl; }
       };
int main()
{ B b1(11,77,99);
  //b1.show_A2(); //B 从 A 类保护继承，所以不能直接访问 A 类的公有成员
  C c1(1,7,9); c1.show_C();;
  //c1.show_A2(); //C 从 B 类公有继承，而 B 从 A 保护继承，所以不能直接访问 A 类的公有成员
  return 0;
}/* 执行结果：show_A1(): i=1 j=7
              show_A2(): i=1 j=7
              show_B():  x=9
                  show_C():           */
```




```
//例 4：派生类的成员和友员访问基类的保护成员
#include<iostream.h>
class A
{ private:   int x;
  protected: int y;
  public:    int z;
    A(int xx,int yy,int zz):x(xx),y(yy),z(zz){}
    void sA()
    { cout<<"A: x="<<x<<" y="<<y<<" z="<<z<<endl; }
    void sA1(A a)
    { cout<<"x="<<x<<"  a.x="<<a.x<<endl; }
    friend void fA(A a);
};
void fA(A a)  //A 的友员函数
{ cout<<"fA(): "<<a.x<<"  ";
  cout<<a.y<<"  ";
  cout<<a.z<<endl;
}

class B1:public A
{ public:
    B1(int xx,int yy,int zz):A(xx,yy,zz){}
    void sB1(B1 b)
    { //cout<<x<<"  ";  //不能访问基类的私有成员
      //cout<<b.x<<"  ";//error
      cout<<y<<"  ";    //可以直接访问从基类公有继承来的保护成员
      cout<<b.y<<"  ";  //可以通过自己类的对象访问从基类公有继承来的保护成员
```

```
      cout<<z<<" ";    //可以直接访问从基类公有继承来的公有成员
      cout<<b.z<<endl;  //可以通过自己类的对象访问从基类公有继承来的公有成员
      sA();             //直接调用基类的公有成员
      B1 bb(41,51,61);  //
      cout<<bb.y<<" "; //类内部创建的本类对象可以访问基类的保护成员
      cout<<bb.z<<endl; //类内部创建的本类对象可以访问基类的公有成员
      A aa(41,51,61);   //
      //cout<<aa.y<<" ";//类内部创建的非本类(基类)对象不能访问基类的保护成员
      cout<<aa.z<<endl; //类内部创建的非本类(基类)对象可以访问基类的公有成员
    }
    void sB2(A b)
    { cout<<y<<" ";
      //cout<<b.y<<" ";//类内部:非本类对象不能访问基类的保护成员
    }
    friend void fB11(B1 b);
    friend void fB12(A a);
};
void fB11(B1 b)
{ //cout<<b.x<<" ";           //派生类的友员不能访问基类的私有成员
  cout<<"fB11(): "<<b.y<<" "; //派生类的友员可以通过该派生类的对象访问基类的保护成员
  cout<<b.z<<endl;            //派生类的友员可以通过该派生类的对象访问基类的公有成员
}
void fB12(A a)
{ //cout<<a.x<<" ";   //派生类的友员不能通过基类的对象访问基类的私有成员
  //cout<<a.y<<" ";   //派生类的友员不能通过基类的对象访问基类的保护成员
  cout<<"fB12(): "<<a.z<<endl; //派生类的友员可以通过基类的对象访问基类的公有成员
}

class B2:protected A
{ public:
    B2(int xx,int yy,int zz):A(xx,yy,zz){}
    void sB2(B2 b)
    { //cout<<x<<" ";  //不能访问基类的私有成员
      //cout<<b.x<<" ";
      cout<<y<<" ";
      cout<<b.y<<" ";
      cout<<z<<" ";
      cout<<b.z<<endl;
      A aa(41,51,61);  //类内部创建的非本类对象不能访问基类的保护成员
      //cout<<aa.y<<" ";//
      cout<<aa.z<<endl;
      sA();
    }
};
class B3:private A
{ public:
    B3(int xx,int yy,int zz):A(xx,yy,zz){}
    void sB3(B3 b)
    { //cout<<x<<" ";  //不能访问基类的私有成员
      //cout<<b.x<<" ";
      cout<<y<<" ";
      cout<<b.y<<" ";
      cout<<z<<" ";
      cout<<b.z<<endl;
```

```
      A aa(41,51,61);  //类内部创建的非本类对象不能访问基类的保护成员
      //cout<<aa.y<<"  ";//
      cout<<aa.z<<endl;
      sA();
    }
};

void main()
{ A a1(44,55,66),a2(444,555,666);        a1.sA1(a2);
  B1 b11(1,2,3),b12(7,8,9);              b11.sB1(b12);
  B2 b21(11,22,33),b22(77,88,99);        b21.sB2(b22);
  B3 b31(111,222,333),b32(777,888,999); b31.sB3(b32);
  //cout<<b11.y<<endl;//类外部创建的对象不能访问基类的保护成员
  fA(a1);  fB11(b11);  fB12(a2);
}/* 执行结果：x=44  a.x=444
             2  8  3  9
             A: x=1 y=2 z=3
             51  61
             61
             22  88  33  99
             61
             A: x=11 y=22 z=33
             222  888  333  999
             61
             A: x=111 y=222 z=333
             fA(): 44  55  66
             fB11(): 2  3
             fB12(): 666         */
```

18 运算符重载 p/396

重载：同名不同义，在不同的情况下可表现出不同的行为。

重载：函数重载
操作符重载－－－一个运算符可进行多种不同含义的解释

操作符重载：使得操作符适用于类的对象之间的操作。提高程序的可读性。

18.1 运算符重载的需要性

问题的提出：在 C++ 中，定义一个类就产生了一个新的类型，类的对象就和变量一样可以作为参数进行传递，也可以作为函数的返回类型

↓

希望某些运算符能够对整个对象进行操作，而不是 C 中的简单操作，例如：+ 运算符能够实现 2 个对象间的加

↓

把某些事交给系统去做，用户只要知道相加就可

↓

提出运算符重载，扩充运算符的功能。即：对运算符进行重载定义，然后使用，由系统自动判断究竟采用哪一个具体的运算符

↓

增强了 C++ 语言的可扩充性

例如：类 A 的对象 a1、a2、a3，希望：a3 = a1 + a2;
即：分别把对象 a1 和 a2 的各个数据成员值对应相加，然后赋给对象 a3

实现途径：操作符重载

18.2 如何重载运算符

重载运算符的实现：运算符函数，即以操作符命名的函数（重载操作符）
操作符重载的函数名规定：由关键字“operator”和操作符组成
例如：类 A 的对象 a1 和 a2 相加，就调用“operator + ”函数

a1+a2 ⟹ operator+(a1,a2)
a1+a2 ⟹ a1.operator+(a2)

类类型的操作符函数可以是成员函数或友元函数。

1. 一元操作符重载的一般形式

⑴重载为成员函数（有 this 指针）
重载声明：
返回类型标识符 operator 被重载的操作符();
重载定义：
返回类型标识符 类名标识符::operator 被重载的操作符(){...}
⑵重载为友元函数（无 this 指针）
重载声明：
friend 返回类型标识符 operator 被重载的操作符(const 类名& 形参名);
重载定义：
返回类型标识符 operator 被重载的操作符(const 类名& 形参名){...}

2. 二元操作符重载的一般形式

⑴重载为成员函数（有 this 指针）
重载声明：
返回类型标识符 operator 被重载的操作符(const 类名& 形参名);
重载定义：
返回类型标识符 类名标识符::operator 被重载的操作符(const 类名& 形参名){...}
⑵重载为友元函数（无 this 指针）
重载声明：
friend 返回类型标识符 operator 被重载的操作符(const 类名& 形参名 1,const 类名& 形参名 2);
重载定义：
返回类型标识符 operator 被重载的操作符(const 类名& 形参名 1,const 类名& 形参名 2){...}

例 1：二元加、一元减操作符重载为成员函数

```
#include<iostream.h>
class A{  int x,y;
      public:
        A(int xx,int yy):x(xx),y(yy){}
        A(){x=0;y=0;}
        A operator+(const A&b)
           { return A(x+b.x,y+b.yy); }
        A operator-()
           { return A(-x,-y); }
        void show()
           {cout<<"x="<<x<<"  y="<<y<<endl;}
      };
void main()
{ A a1(5,55),a2(6,66),a3;
```

```
  a3=a1+a2;   //调用操作符重载函数: a1.oprator +(a2)
  a3.show();
  a1=-a1;     //调用操作符重载函数: a1.operator -()
  a1.show();
}
  /*执行结果:  x=11  y=121
               x=-5  y=-55   */
```

例 2：二元加、一元减操作符重载为友元函数

```
#include<iostream.h>
class A{  int x,y;
        public:
          A(int xx,int yy):x(xx),y(yy){}
          A(){x=0;y=0;}
          friend A operator+(const A&a,const A&b);
          friend A operator-(const A&a);
          void show()
             {cout<<"x="<<x<<"  y="<<y<<endl;}
       };
A operator+(const A&a,const A&b)
  {return A(a.x+b.x,a.y+b.y);}
A operator-(const A&a)
  {return A(-a.x,-a.y);}

void main()
{ A a1(5,55),a2(6,66),a3;
  a3=a1+a2;  //调用操作符重载函数: a1.oprator +(a2)
  a3.show();
  a1=-a1;     //调用操作符重载函数: a1.operator +()
  a1.show();
}
  /*执行结果:  x=11  y=121
               x=-5  y=-55   */
```

有关操作符重载函数的说明：

1. 参数个数有所规定：一元操作符只能有一个参数，二元操作符只能有二个参数；但是成员函数与友元函数的表现形式不同
2. 返回类型可以任意，但是最好不要定义成 void
3. 操作符重载函数的功能应明确，一般不要执行其他不相关的操作。
4. 重载后的操作符的使用语法、优先级、结合性均不变。
5. 如果操作数都是 C++ 的基本数据类型，不允许对操作符进行重载
6. C++ 规定：=、()、[]、-> 这 4 种操作符必须重载成成员函数

18.3 值返回与引用返回

注意：不能对临时对象进行“引用”。如果函数的返回值是个临时对象，则采用“值返回”。

p/401 例题

18.4 运算符重载为成员函数

操作符重载为类的成员函数，上面已介绍过

两种重载形式（成员函数、友元函数）的比较：

1. 一元操作符

   重载成成员函数较好

2.二元操作符

除 =、()、[]、-> 这 4 种操作符必须重载成成员函数外，其他操作符重载成友元函数较好

例，a 是类 A 的一个对象，对于表达式：27.5 + a

对友元函数不存在问题：operator+(A(27.5),a)

但是，对成员函数存在问题：(27.5).operator+(a) //非法：27.5 不是一个对象，无 this 指针

18.5 重载增量运算符

＋＋、－－：是一元操作符

但是，可以是前缀或后缀。所以，重载时有所不同

前缀：一元操作符

后缀：二元操作符

例 1：++和--操作符重载成成员函数

```
#include<iostream.h>
class Counter{  unsigned value;
              public:
                Counter(int v=0){value=v;}
                Counter operator++()
                  { value++;   cout<<"1... "<<value<<"  ";
                    return *this; }
                Counter operator++(int)
                  { Counter t; cout<<"2... "<<value<<"  ";
                    t.value=value++; return t; }
                void display()const
                  { cout<<value<<endl; }
             };
void main()
{ Counter a(11),b(22),c;
  ++a;          //自动调用 operator++()
  a.display();
  c=b++;        //自动调用 operator++(int)
  b.display();
  c.display();
}
  /*执行结果:  1... 12   12
               2... 22   23
               22      */
```

例 2：++和--操作符重载成友元函数

```
#include<iostream.h>
class Counter{  unsigned value;
              public:
                Counter(int v=0){value=v;}
                friend Counter operator++(Counter&);
                friend Counter operator++(Counter&,int);
                void display()const
                  { cout<<value<<endl; }
             };
Counter operator++(Counter&cc)
   { cc.value++;   cout<<"1... "<<cc.value<<"  ";
     return cc; }
Counter operator++(Counter&cc,int)
```

```
   { Counter t; cout<<"2... "<<cc.value<<"  ";
     t.value=cc.value++; return t; }
 void main()
{ Counter a(11),b(22),c;
  ++a;         //自动调用 operator++(Counter&)
  a.display();
  c=b++;       //自动调用 operator++(Counter&, int)
  b.display();
  c.display();
}
  /*执行结果:  1... 12   12
               2... 22   23
               22     */
```

18.6 类型转换运算符

类是用户自定义的类型
两个不同的类间无法进行类型转换，类类型转换指：类与基本类型之间的转换
类类型转换实际上是通过“重载”实现的

类型转换：隐式－－－系统自动完成转换

显式 { 强制转换法
       函数转换法

类的类型转换均属显式、函数转换法

1. 基本数据类型间的转换

隐式转换：例：**int x; float y; y=x;** //y=x; 进行隐式转换
显式转换：①强制转换：例，int x,y; float z; z=(float)(x+y);
②函数转换：例，int x,y; float z; z=float(x+y); //括号不能省

2. 类类型与基本类型之间的转换

隐式转换：利用构造函数－－－－ 完成由其他基本类型向本类型转换的操作
如果无相应的构造函数，则被禁止转换

显式转换：利用类型转换函数－－ 完成由本类型向其他基本类型转换的操作
声明格式：operator 类型标识符();
定义格式：类名::operator 类型标识符()
{ ..... }

说明：
类类型转换函数：无函数返回类型，无参数。因为，总是转换成“类型标识符”所指出的类型，且被转换的总是本类的一个对象（由 this 指针指出）。

```
例：#include<iostream.h>
   class Counter{  int v1;
                   char v2;
                 public:
                   Counter(int v=0,char vv='?'){ v1=v; v2=vv; }
                   operator char(){ return v2; }
                   operator int() { return v1; }
                   void display()
                     { cout<<"Counter: v1="<<v1<<"  v2="<<v2<<endl; }
                  friend Counter fn(Counter c);
                };
    Counter fn(Counter c)
```

```
{ c.v1=c.v1*2; return c; }

 void main()
{ char x; int y;
  Counter c1(64,'%'),c2(99,'@');
  c1.display();  c2.display();
  c2=fn(12345); //实参 12345 通过构造函数 Counter(int v=0,char vv='?')转换成 Counter 类型
  c2.display();
  x=c1;          //通过成员函数 operator char()转换成 char 型
  y=c1;          //通过成员函数 operator int() 转换成 int 型
  cout<<x<<"  "<<y<<endl;
} /*执行结果:  Counter: v1=64  v2=%
               Counter: v1=99  v2=@
               Counter: v1=24690  v2=?
               %  64                    */
```

18.7 赋值运算符

在一般情况下 C++ 允许同类的两个对象互相赋值，例如：

```
class A{ ..... }a1,a2;
a1=a2;   //表示，把对象 a2 的各数据成员的值逐项复制给对象 a1
```

但是，在某些情况下会出现问题，例如“指针悬挂”(p/321，对象创建时存在动态分配内存，析构时就会出现指针悬挂）问题，这时必须对类的赋值运算符进行重载，以解决该问题。另外，如果对象的某个数据是指针指出的不同长度的数组时，也需要考虑赋值运算符重载。 p/411：例题分析

例：

```
#include<iostream.h>
#include<string.h>
class A{  private: char *p;
        public:
          A(char *pp)
            { p=new char[strlen(pp)+1];
              strcpy(p,pp); cout<<"new ..."<<p<<endl; }
          ~A(){ delete p; cout<<"delete p..."<<endl; }
          A& operator=(A& r);  //赋值运算符重载成员函数
        };
A& A::operator=(A&r)
{ if (this==&r) return *this;  //如果进行赋值操作的两个对象是同一个对象，则立即返回
  //delete p; p=new char[strlen(r.p)+1];
  strcpy(p,r.p);
  cout<<"operator =..."<<p<<endl;
  return *this;
}
void main()
{ A a1("ijklmn"),a2("123456789");
  a2=a1; //自动调用自定义的赋值操作函数
}/*执行结果：new...ijklmn
             new...123456789
             operator =...ijklmn
             delete p...
             delete p...          */
```

关于该例题的说明：

如果不定义赋值运算符重载成员函数：则此程序编译通不过。

如果定义了赋值运算符重载成员函数，并且在该函数体中无语句：delete p; p=new char[strlen(r.p)+1]; 本程序能够通过编译，并且被执行。但是，如果把主函数中的语句 a2=a1; 改成：a1=a2; 执行时就出错，程序被中断执行；如果在赋值重载函数中加上语句 delete p; p=new

char[strlen(r.p)+1]；则就完全正确。所以，遇到这种情况应该编写赋值重载成员函数

19 I/O 流

流：数据的流动，抽象为“流”类

C++ 的流类库提供了一组“类”，利用这些类可以实现计算机与外设之间进行数据交流。

19.1 printf 和 scanf 的缺陷

非类型安全：编译系统对函数原型进行检查（参数个数、参数类型），可以避免许多错误，但是编译系统对 printf() 和 scanf() 函数只检查第一个参数（参数个数和类型信息包含在第一个参数中），所以，其他的错误无法在编译时被发现。

不可扩充性：不能对类对象进行输入输出。

19.2 I/O 标准流类

读取：从“流”中获取数据

写入：向“流”中添加数据

iostream.h：标准 I/O 流头文件，在该文件中对各个流类定义了各自的全局对象。

cin： C++ 提供的标准输入流对象名，对应设备是键盘

cout：C++ 提供的标准输出流对象名，对应设备是屏幕

cerr：C++ 提供的标准错误流对象名，对应设备是屏幕

clog：C++ 提供的标准打印流对象名，对应设备是打印机

例如，cin 是 istream 流类的对象，创建该对象时，传递给构造函数的实参是“标准输入设备（键盘）”，即该类的一个数据成员是“设备名”，对象 cin 的“设备名”是键盘；“>>”是 istream 类的操作符重载函数，执行“>>”操作就从键盘上读取一个数据给指定的变量（函数的另一个参数）。

p/416 “原理”

例：p/417 的例题

```
#include<iostream.h>
void fn(int a,int b)
{ if(b==0)
     cerr<<"zero encountered. "<<"The message cannot be redirected";
  else
  cout<<a/b<<endl;
}
void main()
{ fn(20,2);
  fn(20,0);
}
/*执行结果：10
            zero encountered. "<<"The message cannot be redirected      */
```

19.3 文件流类

fstream.h：“文件流”头文件，在该文件中对各个文件流类进行了定义，但是没有预定义各自的全局对象。因为，在 C++ 中文件类不是标准设备，不能预先定义。

在头文件 fstream.h 中给出了 fstreambase、ifstream、ofstream、fstream 类的定义

fstreambase 是 ifstream 和 ofstream 的公共基类，fstream 是 ifstream 和 ofstream 的派生类，都是公有继承关系。文件流类的操作（成员函数）应用于外部设备，最典型的、最常用的是“磁盘文件”。

要进行文件操作，必须创建文件流对象，创建文件流对象时必须规定文件名和文件的打开方式（文件名和文件的打开方式是传递给构造函数的实参）

p/418：构造函数，文件打开方式，文件保护方式

1. 打开文件

只有打开文件，才可以访问文件（对文件流的对象进行操作）

通过创建 ifstream、ofstream、fstream 的对象，在创建时自动调用各自的带参数的构造函数来打开文件；或者，对已有的流对象用 open()函数来打开文件。

创建 ifstream 的对象：缺省文件打开方式为“in”

创建 ofstream 的对象：缺省文件打开方式为“out”

创建 fstream 的对象： 无缺省文件打开方式，必须指定方式

2. 关闭文件

1. 调用公共基类 fstreambase 的 close()成员函数

注意：文件被关闭了，但是“流”对象仍然存在。可以利用这个流对象再打开别的文件，对别的文件进行操作

2.流对象生命周期结束时，系统自动调用析构函数关闭该对象所操作的文件，并析构对象

例：

```
#include<iostream.h>
#include<fstream.h>
void main()
{ int i;
  double x;
  char a[10],b[10];
  ofstream os1,os2("abc2.dat");//os1 是个 ofstream 类的没有初始值的对象(没有打开文件)
  os1.open("abc1.dat");          //对象 os1 调用成员函数 open()以缺省方式打开一个文件
  os1<<"shang hai 123"<<endl;  //向文件 abc1.dat 中写入一行数据
  os2<<99.12<<endl<<"jiao_tong"<<endl; //向文件 abc2.dat 中写入二行数据
  os1.close();
  os2.close();
  ifstream is1;
  is1.open("abc1.dat");
  is1>>a>>b>>i;
  cout<<a<<" "<<b<<"   "<<(i*2)<<endl;
  is1.close();                                  //关闭文件，但对象 is1 依然存在
  is1.open("abc2.dat");
  is1>>x>>a;
  cout<<x<<"   "<<a<<endl;
  os1.close();
  fstream fs("abc1.dat",ios::in|ios::out); //以既可读又可写的方式打开文件
  fs>>a>>b>>i;                                  //读完后文件指针指向下一个
  cout<<"fs: "<<a<<" "<<b<<"   "<<i<<endl;
  fs<<"kkkkk ppp "<<777<<endl;                  //向文件 abc1.dat 中再写入一行数据
  fs.close();
  fs.open("abc1.dat",ios::in);                  //以只读方式打开文件
  fs>>a>>b>>i;
  cout<<"fs: "<<a<<" "<<b<<"   "<<i<<endl;
  fs>>a>>b>>i;
  cout<<"fs: "<<a<<" "<<b<<"   "<<i<<endl;
  fs.close();
} /*执行结果：  shang hai   246
                99.12   jiao_tong
                fs: shang hai   123
                fs: shang hai   123
```

```
                fs: kkkkk ppp    777     */
```

说明：1.写入时以输入回车符标记一行结束，读取时以空格或回车符分割数据单元
2.如果构造函数或 open()函数中指定的文件建立在该程序的可执行文件所在的目录中，文件路径可省，否则必须指定文件路径(路径分隔符用两个反斜杠：\\)

19.4 串流类

C++ 允许把字符串当作设备，进行对串流类对象的输入输出操作。
strstrea.h 头文件：包含串流类 ostrstream、istrstream、strstream 的定义
在 C++ 中串流类不是标准设备，不能预先定义

例：p/420 的例题

```
#include<iostream.h>
#include<strstrea.h>
char *parseString(char* pString)
{ istrstream inp(pString,0);
  int aNumber;
  float balance;
  inp>>aNumber>>balance;
  char* pBuffer=new char[128];
  ostrstream outp(pBuffer,128);
  outp<<"a number= "<<aNumber<<", balance= "<<balance<<'\0'; //必须加'\0'，否则输出不正确
  return pBuffer;
}
void main()
{ char *str="1234 100.35";
  char *pBuf=parseString(str);
  cout<<pBuf<<endl;
  delete []pBuf;
} /*执行结果： a Number= 1234, balance= 100.35  */
```

19.5 控制符

p/421

19.6 使用 I/O 流类的成员函数

1.用 getline()读取输入行
2.用 get()读取一个字符
3.用 get()输入一系列字符
4.输出一个字符

19.7 重载插入运算符

C++提供的插入运算符：<<，是重载运算符而成的
用户可以再重载插入运算符，以完成特定的输出。

p/428 例题讲解:

```
#include <iostream.h>
#include <iomanip.h>  //包含控制符对象的头文件：p/422

class RMB
{  public:
    RMB(double v=0.0)
      { yuan=v;  jf=(v-yuan)*100.0+.5; }
    operator double()          //类型转换函数
      { return yuan+jf/100.0; }
```

```
    void display(ostream&out)
      { out<<yuan<<'.'<<setfill('0')<<setw(2)<<jf<<setfill(' '); }
  protected:
      unsigned int yuan;
      unsigned int jf;
};

ostream&operator<<(ostream&oo,RMB&d)  //一般操作符重载函数
{ d.display(oo);  return oo; }

void main()
{ RMB rmb(1.5);
  cout<<"Initially rmb="<<rmb<<"\n";
  /* cout<<"Initially rmb="操作用的是 iostream 类的<<重载函数，返回
     cout 对象，然后执行 cout<<rmb 操作，用的是上面定义的重载函数 */
  rmb=2.0*rmb;
  /* 把对象 rmb 利用类型转换函数转换成 double 类型的对象，然后
     与 2.0 相乘，再利用构造函数再把乘积转换成 RMB 类型的对象 */
  cout<<"then rmb="<<rmb<<'\n';
}
```

19.8 插入运算符与虚函数

19.9 文件操作

文件输出：写文件　　p/433 例题讲解
文件输入：读文件　　p/435 例题讲解

20 模板

模板：C++ 支持参数化多态性的工具
参数化多态性：对程序中所处理的对象的类型参数化

20.1 模板的概念

C++ 的主要构件是类和函数
C++ 模板：函数模板、类模板

1. 函数模板和模板函数

函数模板：
定义一个函数时，必须明确函数的返回类型、参数的类型，这些类型如果被参数化，则该函数的应用就广。参数化的函数称为函数模板。

例如：
```
int abs(int x)
    { return x<0?-x:x; }
double abs(double x)
    { return x<0?-x:x; }     //必须写多个函数
```

改成：
```
T abs(T x)
    { return x<0?-x:x; }     //写一个函数模板
```

模板函数：
函数模板是生成函数代码的样板，当参数类型确定后，编译时用函数模板生成一个具有确定类型的函数，这个由函数模板而生成的函数称为模板函数。

例如：
```
#include<iostream.h>
template<typename T>
T abs(T x)
```

```
  { return x<0?-x:x; }
void main()
{ int n=-5;
  double d=-5.5;
  cout<<abs(n)<<endl;  //编译时生成 int abs(int x){...} 模板函数
  cout<<abs(d)<<endl;  //编译时生成 double abs(double x){...} 模板函数
} /*执行结果：  5
                5.5  */
```

2. 类模板和模板类

类模板：

定义一个类时，必须定义各成员的类型，这些类型如果被参数化，则该类的应用就广，可以生成一组类。参数化的类称为类模板。

例如：
```
class A{  int x;
          public:
            A(int xx):x(xx){}
            int fn(){return x*x*x; }
          };
class B{  double x;
          public:
            B(double xx):x(xx){}
            double fn(){return x*x*x; }
          };
```
改成：
```
class A{  T x;
          public:
            A(T xx):x(xx){}
            T fn(){return x*x*x; }
          };
```

模板类：

类模板是生成类的样板，当类的参数类型确定后，编译时用类模板把类型参数换成确定的类型，就生成一个具有确定类型的类，这个由类模板而生成的类称为模板类。

例如：
```
#include<iostream.h>
template<typename T>
class A{  T x;
          public:
            A(T xx):x(xx){}
            T fn(){return x*x*x; }
          };
void main()
{ A<int>a1(5);           //编译时生成一个把 T 转换成 int 的模板类
  A<double>a2(5.5);      //编译时生成一个把 T 转换成 double 的模板类
  cout<<a1.fn()<<"  "<<a2.fn()<<endl;
} /*执行结果：  125  166.375  */
```

20.2 为什么要用模板

目的：避免重复编程，增加程序的灵活性

20.3 函数模板

声明和定义：

格式：

template<typename 参数化类型名 1, typename 参数化类型名 2, … , typename 参数化类型名 n>

函数返回类型 函数名(形参列表)

```
    { … }
```

说明：

一旦声明了某个参数化类型名，该类型名就可以在函数定义中被使用

例：建立一个能够处理多种类型的两个数的乘法运算的函数模板

```
#include<iostream.h>
template<typename T1,typename T2>  //声明紧接在下面的函数 cal()是个函数模板
T1 cal(T1 x,T2 y)                  //该函数模板有两个参数化的类型名：T1 和 T2
  { return T1(x*y); }
void main()
{ int x=11;
  double y=99.99;
  cout<<cal(x,y)<<endl; //分别以实参 x,y 的类型 int 和 float 替换参数化类型名 T1 和 T2
  cout<<cal(y,x)<<endl; //分别以实参 y,x 的类型 float 和 int 替换参数化类型名 T1 和 T2
} /*执行结果： 1099
               1099.89    */
```

20.4 重载函数模板

可以重载函数模板，即函数名可以与函数模板名同名。

```
例：#include<iostream.h>
    template<typename T>
    void f(T*p)
      { p->f(); cout<<"template......"<<endl; }
    void f(int p)
      { cout<<"f()... "<<p<<endl; }
    class A{public:
              virtual void f()
                { cout<<"A..."<<endl; };
           };
    class B:public A
           {  int x;
            public:
              void f()
                { cout<<"B..."<<endl; }
            };
    void main()
    { A *pa=new B;
      f(5566);
      f(pa);
    } /*执行结果：f()... 5566
                  B...
                  template...   */
```

说明：系统先匹配重载函数，如果匹配不成功，再匹配函数模板

20.5 类模板的定义

格式：

```
template<typename 参数化类型名 1, typename 参数化类型名 2, … , typename 参数化类型名 n>
class 类名{ …
            …
          };
```

对于非内联的成员函数的定义格式：

```
template<typename 参数化类型名 1, typename 参数化类型名 2, … , typename 参数化类型名 n>
类名<参数化类型名 1, 参数化类型名 2, … , 参数化类型名 n>::成员函数名(参数列表)
```

{ … }

p/443～445 例题讲解

20.6 使用类模板

例：#include<iostream.h>

```
    template<typename T,int N> //声明紧接在下面的类 A 是个类模板
    class A{  T a[N];
            public:
              A();
              void show()
                { cout<<"N= "<<N<<endl; }
              void sh();
            };
    template<typename T,int N>
    A<T,N>::A()
      { for(int i=0;i<N;++i)
          { cout<<"input a["<<i<<"] : ";
            cin>>a[i]; }
      }
    template<typename T,int N>
    void A<T,N>::sh()
      { for(int i=0;i<N;++i)
          { cout<<"a["<<i<<"]= "<<a[i]<<endl; }
      }
    void main()
    { A<int,5>aa;//用类模板 A 建立一个模板类的对象 aa：用类型 int 替换参数化类型 T，5 替换 N
      aa.show();
      aa.sh();
    } //执行时输入 5 个整数（N 为 5）
```

说明：类模板 A 中的 T 为参数化了的类型名；N 为参数化了的常量，该常量的类型为 int 型

注意：在定义对象时必须加以实参类型说明，上例中的：A<int,5>aa;

20.7 使用标准模板类库：Josephus 问题

p/447

21 异常处理

21.1 异常的概念

异常：指程序运行时出现了非常事件，系统不知道如何处理，就中断执行。例如除数为 0。
异常处理：希望引发异常时，能够解决异常，使得程序能够正常执行

C++中，利用 try、throw、catch 语句进行异常处理
异常是对所有能够预料的运行错误进行处理的机制，使得对运行异常加以控制

21.2 异常的基本思想

在被调用函数中“引发”异常。即如果在被调用函数的执行过程中出现异常，就把异常引发给调用函数，并且立即返回。

在调用函数中“捕捉”异常。即调用函数如果收到被调用函数引发的异常，就执行相应的程序段处理这个异常。

21.3 异常的实现

p/453 ：系统调用 main()，对其中 main()的参数捕捉异常并处理异常

异常处理的一般过程：

1.对可能在运行时会出现异常的地方通过 throw 语句“引发”该异常：

throw 表达式；

说明：①表达式的类型被称为“异常”类型，异常处理时就根据捕捉到的异常，按其类型不同而分别处理

②只有在可能出现异常的地方写上 throw 语句，才会引发异常，否则程序异常中断

③对于函数调用，如果被调用函数引发了异常，返回时不是回到调用点，而是返回到调用函数所指定的地方。

2.通过 try、catch 语句捕捉异常并处理异常：

```
try{...
    ...        语句段
    ...
    }
catch(异常类型声明 1){...
                          ...        异常处理语句段 1
                          ...
                          }
catch(异常类型声明 2){...
                          ...        异常处理语句段 2
                          ...
                          }
        ...
        ...
catch(异常类型声明 n){...
                          ...        异常处理语句段 n
                          ...
                          }
```

其中，“异常类型声明”为估计会引起异常的操作数的类型，其声明语法为：数据类型 标识符（或者：数据类型& 标识符）

例：

```
#include<iostream.h>
double div(double a,double b)
{ if (b==0.)
    throw b;
  return a/b;
}
void main()
{ try{ cout<<"7.3/2.0="<<div(7.3,2.0)<<endl;
       cout<<"7.3/0.0="<<div(7.3,0.0)<<endl;
       cout<<"7.3/1.0="<<div(7.3,1.0)<<endl;
     }
  catch(double)
    { cout<<"except of deviding zero.\n"; }
  cout<<"That is ok.\n";
} /*执行结果：7.3/2.0=3.65
              except of deviding zero.
              That is ok.                */
```

21.4 异常的规则

规则：

1.try 和 catch 是一个整体性的语句，在 try 段中捕捉异常，在 catch 段中处理异常。可以有多个 catch 段，即具有多个处理不同异常类型的程序段。catch 段必须紧跟在 try 段之后。

2.catch(异常类型声明)，“异常类型”必须有，捕捉是通过对数据类型进行匹配实现的

3.如果通过 throw 抛掷了一个异常，被捕捉到后，在本函数中找不到与异常类型匹配的 catch 段，就把控制返回到调用函数（父函数）中，使用相同的规则在父函数中寻找相应的处理程序，...，直到异常被处理，或者被 C++运行系统处理（即异常终止）。

例：
```
#include<iostream.h>
int fn2(int a,int b)
{ if (b==0)
    throw b;
  cout<<"fn2()..."<<endl;  //一旦抛掷“异常”，此句及以下语句均不被执行
  return a/b;
}
int fn1(int a,int b)
{ int z;
  try{ cout<<"fn1()\n"; z=fn2(a,b); }
  catch(double){ cout<<"fn1()   double\n"; }//一旦捕捉到异常，就寻找异常处理程序
  cout<<"fn1()..."<<endl;
  return z;
}
void main()
{ int x=88,y=0;
  try{ cout<<"x/y= "<<fn1(x,y)<<endl; }
  catch(int){ cout<<"main():int  b=0\n"; }  //此异常处理被执行
  cout<<"main()..."<<endl;
} /*执行结果：fn1()
              main():int  b=0
              main()...        */
```

4.异常处理完毕，立即执行处理本异常程序段所在的那个 try 语句后的语句。

21.5 多路捕捉

多路捕捉：一个程序中的所有类型的异常，都会被捕捉。用户定义的“抛掷——捕捉——处理异常”构成一个异常区域，各个异常区域内发生的异常被本区域捕捉，然后处理，如果本区域内没有该异常类型的处理程序段，就由系统捕捉并处理。如果异常不是发生在用户定义的异常区域内，则由系统捕捉并处理。被系统捕捉的异常，均采用“中断”执行来处理。

p/458 例题讲解

21.6 异常处理机制

异常处理机制：

1. 在一个函数“调用链”中，如果链中的某个函数通过 throw 抛掷了一个异常，被捕捉到后，在本函数中找不到与异常类型匹配的 catch 段，就把控制返回到调用函数（父函数）中，使用相同的规则在父函数中寻找相应的处理程序，...，直到异常被处理，或者被 C++运行系统处理（即异常终止）。异常处理完毕，立即执行处理本异常程序段所在的那个 try 语句后的语句。
2. 如果 throw 没有带参数，则表示把捕捉到的“异常”再沿“链”向上抛掷出去。
3. 如果出现 catch(...){}形式，即参数为省略号，是一个与任何类型都匹配的默认异常处理程序。“默认”异常处理程序段应该写在最后面，否则编译通不过。

例：
```
#include<iostream.h>
int fn2(int a,int b)
{ if (b==0)
    throw b;
  cout<<"fn2()..."<<endl;
  return a/b;
}
int fn1(int a,int b)
{ int z;
```

```
    try{ cout<<"fn1()\n"; z=fn2(a,b); }
    catch(double){ cout<<"fn1()   double\n"; }
    cout<<"fn1()..."<<endl;
    return z;
}
void main()
{ int x=88,y=0;
  try{
  //throw;  //如果写此语句，则把异常抛掷给 C++ 运行系统，将中断运行
  cout<<"x/y= "<<fn1(x,y)<<endl; }
  catch(...){ cout<<"main(): default  b=0\n"; }
  cout<<"main()..."<<endl;
} /*执行结果：fn1()
              main():default  b=0
              main()...        */
```

21.7 使用异常的方法

异常族系：由多个异常组成的族系
组成异常族系的两种方法：异常枚举族系、异常派生层次结构 （p/464 例题）

例：

```
#include<iostream.h>
class A{public:
          int x,y;
          A(int x1,int y1)
            { x=x1; y=y1; }
          void show(){ cout<<"A:: x="<<x<<" y="<<y<<"  "; }
       };
class B:public A
       {  int x,y;
        public:
          B(int x1,int y1,int x2,int y2):A(x1,y1),x(x2),y(y2){}
          void show(){ A::show(); cout<<"B:: x="<<x<<" y="<<y<<endl; }
          void div();
       };
void B::div()
          { if(B::y==0)
            throw *this;  //引发基类型 B 的"异常"
            if(A::y==0)
            throw A(x,y);  //引发基类型 A 的"异常"
          }

void main()
{ B b(101,0,202,9);
  b.show();
  try                                          //捕捉异常
    { b.div(); }                               //该程序运行到此时，捕捉到异常
  catch(B){ cout<<"B:: Divider is 0"<<endl; }  //处理 B 类型的异常
  catch(A){ cout<<"A:: Divider is 0"<<endl; }  //处理 A 类型的异常
} /*执行结果：A:: x=101 y=0  B:: x=202 y=9
              A:: Divider is 0              */
```

说明：1.在程序执行过程中，一旦发生"异常"，立即返回到"调用者"

2.该程序在执行函数 B::div()时发生了一个 A 类型的"异常"，尽管是 B 类型对象调用的，但是捕捉到的还是 A 类型异常，故由 A 类型异常处理程序段处理

3.try 与 catch 组成一个语句，不能在一个函数内只有 try 而没有 catch，或只有 catch 而没有

try 的声明

4. 如果把 main()函数中的语句 B b(101,0,202,9)；改成 B b(101,9,202,0)； 则由 B 类型异常处理
5. 基类型的异常可以处理派生类型的异常。如果把 catch(A)程序段写在 catch(B)的前面，则不管 A 类型异常还是 B 类型异常，总是执行 catch(A)