目　录

第 1 章　引　论

一、基本内容

机器语言、汇编语言、高级语言、元语言、编译连接、早期绑定、晚期绑定、面向对象的设计步骤、封装、重载、多态、继承、抽象、程序结构与组织。本章对面向对象的有关概念不要求完全掌握。

二、学习要点

机器语言是一种计算机自身可以识别的语言，机器语言程序是由机器指令和数据形成的二进制文档。机器语言指令通常由操作码和地址码构成。机器语言程序难于理解、编程繁琐、不易维护。

汇编语言是一种符号化的能直接将汇编指令翻译为机器指令的程序设计语言。汇编语言和机器语言同为低级语言。汇编语言更易理解和用于编程，但它仍然是一种面向机器的低级程序设计语言。

高级语言程序是一种以类似于自然语言形式描述指令及数据的程序设计语言。高级语言程序经编译可生成低级语言程序。高级语言通常指是面向过程的程序设计语言，即描述的是问题求解的过程、算法或方法，问题求解的常用手段是功能分解。面向对象的程序设计语言是一种能对新类型描述其属性和操作的高级语言，同时为新类型的定义提供了重载、封装、多态、继承、组成等描述机制。

元语言是一种用来描述其他语言的语言。其特点是描述准确严格、抽象程度高。常用的描述程序设计语言的元语言是 BNF 语言。

编译程序是用来将高级语言翻译为低级语言的程序，反编译程序是用来将低级语言翻译为高级语言的程序。程序设计语言的翻译方式通常有两种，即解释方式和编译方式。C++一般采用编译方式。

高级语言程序通常要经过预处理、词法分析、语法分析、代码生成和模块连接等环节，才能被编译成可被计算机执行的程序。

绑定(binding)就是将函数的入口地址同函数调用指令的地址码相联系的过程。绑定就是要计算被调用函数的入口地址，并将该地址存放到函数调用指令的地址码部分。绑定分为静态绑定和动态绑定两种形式，通常又叫早期绑定和晚期绑定。静态绑定在程序执行前完成，由编译程序或操作系统的装入程序计算函数的入口地址；动态绑定则在程序执行过程中完成，由应用程序自身计算函数的入口地址。

面向对象的技术八十年代初开始出现，SMALLTALK、C++、JAVA、C#等是较为著名的面向对象的程序设计语言。C++在 20 世纪 80 年代初出现，直到 1983 年 C++的名字才正式确定下来。1986 年，Bjarne Stroustrup 在美国 AT&T 的贝尔实验室开发了 C++。

面向对象的设计要经历系统分析、系统设计、对象设计与对象实现等四个阶段，每个阶段之间的界限不是十分明显。这种早期模型和晚期模型的一致性，使面向对象的技术能更

好地支持快速原型法。

封装是将对象的“属性”和“方法”包装在一起、并对外部提供不同权限的访问接口的机制。重载是指用同名的操作针对不同的类型完成不同的功能。重载是多态的一种特例，重载又称为编译时多态，多态则特指运行时多态。多态是指同名的操作针对不同的类型表现出不同的行为。继承是指一种类型接受并利用另一种类型的属性和操作的机制。继承可以分为取代继承、包含继承、受限继承和异化继承等多种方式。

抽象是指一种从事物实例得到事物的共同属性和操作从而形成事物类型描述的过程，或从若干低级的事物类型不但提炼形成高级或更具普遍意义事物类型描述的过程。

C++程序是一种由变量、过程或函数以及类型描述构成的混和结构的程序织。因此，从这个意义上说，C++不是一种纯的面向对象的程序设计语言。

三、习题答案

1.1 高级语言程序经过哪些阶段才能成为可执行程序？

解：高级语言程序通常要经过预处理、词法分析、语法分析、代码生成和模块连接等阶段，才能被编译成可被计算机执行的程序。

1.2 什么叫静态连接？什么叫动态连接？它们有什么区别？

解：静态连接是由编译程序完成的。编译程序将编译生成的目标语言连接成可执行程序文件的过程。

动态连接是由操作系统完成的。在应用程序执的行过程中，操作系统根据被调用的函数名连接内存中的动态连接库函数。若相关动态连接库被移出内存，则重新装入动态连接库并重新连接。

静态连接是在程序执行之前完成的，动态连接是在程序的执行过程中完成的。静态连接程序在执行时不需要再次装入程序文件，动态连接程序在执行时一般需要重新装入程序文件，因此，静态连接程序一般来说比动态连接程序执行速度快。静态连接程序需要将程序文件全部装入内存，如果不同的程序包含了同样的函数代码，就会在内存装入这些函数的多个副本，动态连接程序总是共用同一个函数副本，因此，静态连接程序一般要比动态连接程序占用更多的内存。

1.3 什么叫静态绑定？什么叫动态绑定？它们有什么区别？

解：静态绑定是在程序运行之前进行的。编译程序或操作系统在装入程序后，计算函数的入口地址，并将该地址填写到相应的函数调用处。

动态绑定是程序运行之中完成的。应用程序在执行过程中，激活由编译程序生成的、通过静态或动态方式连接的一段代码，计算函数的入口地址并填写到相应的函数调用处。

两者的区别在于：静态绑定在程序执行之前完成，由编译程序或操作系统计算函数的入口地址；动态绑定在程序运行之中完成，由应用程序自己计算函数的入口地址。

1.4 什么叫封装？提供封装机制有什么好处？

解：封装是将对象的“属性”和“方法”包装在一起、并对外部提供不同权限的访问接口的机制。

提供封装机制的好处在于：（1）通过封装对象的“属性”和“方法”，为对象定义了系统边界。（2）在保持封装接口不变的情况下，可以改变对象的内部结构，而不会影响对

象的外部特性，从而为对象进化提供了方便。（3）由于封装屏蔽了对象“方法”的细节，从而保证核心算法不被泄露，有助于保护软件开发机构的知识产权。

1.5 什么叫包含继承？什么叫受限继承？

解：包含继承是一种派生类对象完整继承所有基类“属性”和“操作”、并且增加自己的“属性”和“操作”的继承方法。

受限继承是一种派生类对象部分继承基类“属性”和“操作”、并且不增加新的“属性”和“操作”的继承方法。

1.6 C++语言有何特点？

解：（1）C++是C语言的超集，继承了C语言的代码质量高、运行速度快、可移植性好等特点。（2）C++是一种强类型的语言，这使得开发人员在编译阶段就能发现C++程序的潜在错误。（3）C++的表达能力由于多继承特性、丰富的运算符及运算符重载机制而远远强于其他面向对象的语言。（4）C++通过函数模板和类模板提供了更高级别的抽象能力，从而进一步提高了C++的表达效率。（5）C++提供了面向对象的异常处理机制，从而使程序更加易于理解和维护，并为局部对象提供了自动析构等有效手段，从而可避免因局部对象未析够而造成的资源泄露（包括内存泄露）。（6）C++的名字空间解决了不同机构的软件模块的标示符同名冲突问题，从而为大型软件的开发和软件容错提供了有效手段。（7）在对象的内存管理方面，C++提供了自动回收和人工回收两种方式。这是开发高效率的系统软件所必需的，但另一方面却容易出错且难于掌握。（8）C++程序是由类、变量和模块混合构成，不象 JAVA 那样完全由类构成。

1.7 应当怎样组织C++的程序？

解：一个程序由包含文件、类程序文件、函数程序文件等多种文件构成。

根据程序所用的库函数，收集要包含的标准头文件，将其加入到自己定义的包含文件中。自定义的包含文件供所有程序文件使用，该文件包含类型、变量、函数的说明信息而非定义信息。

类程序文件是指包含了成员函数函数体定义的文件，而不是仅包含成员函数原型的说明信息文件；函数程序文件是指包含了函数头及其函数体定义的文件，而不是仅包含函数原型的说明信息文件。

如果变量、函数不为整个程序所共享，则将它们定义为局部变量和函数。

第 2 章　C++的变量、类型及函数

一、基本内容

声明、定义、左值、右值、变量定义、匿名变量、只读变量、挥发变量、类型解释、枚举类型、引用类型、函数原型、函数重载、编译换名、缺省参数、省略参数、调用开销、函数内联。

二、学习要点

声明是指对变量、函数和类型的不完整描述，定义是指对变量、函数和类型的完整描述。一个变量、函数和类型标识通常可以声明多次，但只能定义一次。变量声明通常只描述变量的类型和名称，但不分配内存或进行初始化。函数声明只说明函数的函数名、参数类型和返回类型，不说明函数的函数体。类型声明通常只说明类型名不描述类型体，主要指 class、struct、union 等类类型，例如，类名的前向引用声明。

C++的变量定义比 C 自由。C 的变量定义必须出现在函数前或语句前等固定位置，而变量定义的初始化很少能用任意表达式。C++ 的变量定义位置几乎没有限制，变量定义的初始化时可以用任意表达式。任意表达式指左值或右值表达式。

左值表达式是指能出现在等号左边的表达式。大部分变量都能出现在等号左边，因而单个变量通常可以做左值表达式；函数调用如果返回值引用类型，则通常也可以出现在等号左边，因此，返回引用类型的函数调用也可作左值表达式。此外，C++的前置++和—运算和赋值运算表达式也是左值表达式。当然，运用运算符重载技术，几乎任何包含对象运算的表达式都可定义为左值表达式。例如，对象加法运算表达式是右值表达式，但可以重载为左值表达式，只是习惯上不能这样做。

右值表达式是指能出现在等号右边的表达式。所有的表达式都是右值表达式，包括单个变量、函数调用和由运算符构成的任何表达式。因此，左值表达式也是右值表达式，即出现在等号左边的表达式一定能出现在等号右边。反之，右值表达式不一定是左值表达式。

C++的变量定义虽然比 C 自由，但在定义自动变量时尤其自动数组变量时，最好按照 C 的定义方法定义，否则可能付出较大的空间代价。

全局变量的初始化通常由开工函数完成，C 程序没有开工函数和收工函数。一个 C++程序编译通常由三个部分构成：开工函数、main 函数、收工函数。因此，即使 main 函数函数体为空，开工函数或收工函数也可能运行并产生输出结果。开工函数和收工函数是由编译程序生成的匿名函数，因而是不可访问或调用的。当应用程序开始运行时，首先执行开工函数、然后执行 main 函数、最后执行收工函数。开工函数主要完成全局变量的初始化和全局对象的构造，收工函数主要负责全局对象的析构。

在对变量进行初始化时，如果初始化表达式的类型与变量的类型不符，则编译程序通常会产生匿名变量。匿名变量由于没有变量名，其值当然是不可访问的。

只读变量用 const 说明，必须在定义的同时立即初始化，之后不能改变只读变量的值。

函数的参数和返回类型也可以是只读的。只读参数的初始化在函数调用时传递值，如果实参的类型和形参类型不同，同样也可能产生匿名变量。在一般情况下，函数的返回类型缺省是只读的，例如 int f( )就相当于 const int f( )。需要注意的是，int &g 不等价于 const int&g( )，前者的函数调用是一个左值，即可以出现这样的表达式 g( )=4，而后者的函数调用是一个右值。

挥发变量用 volatile 说明，多用在多任务环境做信号灯，以便进行 P-V 操作之类的协同处理。挥发变量是指本程序、本进程或本线程没有修改其值但其值在自主变化的变量，出现这样的情况显然是因为另外的程序、进程或线程在操作。

在进行类型解释时，必须按照运算符的优先级和结合性解释，即首先解释优先级较高的运算符，在优先级相同时按结合性规定的顺序解释。在定义类型时，用到的运算符包括星号*、括号( )、数组下标[ ]以及以后还要介绍的成员指针运算符。显然，函数原型 int (f)(char)等价于函数原型 int f(char)。

枚举类型用 enum 说明，在编译后通常翻译为整型，因此，枚举类型也是简单类型。在缺省情况下，枚举元素的整型值通常从零开始，下一个元素依次增 1。枚举元素不能重复，其整型值也可以设定，且整型值可以重复。例如，enum WEEKDAY {Sun, Mon=1, Tue, Wed, Thu, Fri, Sat}。

引用类型用&说明，在编译后翻译为指针。逻辑上，引用变量并不分配存储单元存储变量的值，每次对引用变量的操作都转化为被引用变量的操作。引用变量在定义时必须立即初始化。左值引用变量必须用左值表达式初始化，右值引用变量可用左值或右值表达式初始化，如果初始化表达式的类型和引用变量不符，则可能先产生类型相符的匿名变量，然后用初始化表达式初始化匿名变量，并用匿名变量初始化引用变量。函数参数和返回类型都可以定义为引用类型，函数调用传递实参初始化引用参数时也可能产生匿名变量。右值引用变量是指 const 类型的引用变量，例如，const int &x=3。

函数原型用于描述函数名、函数参数及其类型以及函数的返回类型。C++不允许出现函数名及参数类型一一对应相同而仅返回类型不同的函数原型。函数名相同而函数原型不同的函数称为重载函数。

C++编译程序允许使用 overload 定义重载函数，但现在的编译程序一般都能自动识别重载函数，不需要使用 overload 定义重载函数。编译程序在将变量名和函数名编译为低级语言标识符时，采用了与 C 编译程序不同的名称转换策略。因此，如果不采用 extern "C" int x 形式的说明，就无法和 C 程序连接并访问其变量或函数。

根据 C++编译程序的换名策略，函数名在转换为低级语言函数名时，是由 C++函数名外加参数类型等信息构成的，这种策略巧妙地解决函数重载及重载函数的调用问题，传递不同类型的实参将导致直接调用低级语言相应的函数。

函数可以定义缺省参数，即调用时若不传递实参，将使用参数缺省值作为实参。函数也可以定义省略参数，省略参数用“…”表示，“…”表示 0 至任意个任意类型的参数。在 stdio.h 中，定义了两个典型的省略参数的函数。其中 printf 的函数原型为：int printf(const char*,…)，表示 printf 可以打印任意个数的值。

在调用一个函数，通常要通过压栈来传递参数的值，调用完毕要清除栈顶的内容以保持栈指针在调用前后是相同的。因此，除非调用嵌套的层次无比深，或者出现了无穷递归调

用，否则，用高级语言编写的程序是不会轻易出现栈溢出的。其实，用汇编语言编写程序，也可采用这种保持调用前后栈平衡的策略。

调用时的压栈出栈不是函数体所定义的有效计算，这些有效计算之外的额外指令称为调用开销。当函数体很小时，调用开销就显得非常之大。函数内联可以消除调用开销，提高程序的执行效率。通过直接将函数体所定义的计算指令插入到函数调用处，函数内联避免了函数调用从而消除了调用开销。由此可见，当函数体较长时，内联将会大大增加程序的长度，反而不会有什么好处。

函数内联用 inline 说明，但并是加上 inline 就能内联成功。内联成功的前提是：(1) 内联函数的定义必须出现在函数定义之前；（2）函数体不能出现 if、switch、for、while、do、？：和函数调用等分枝转移类型的语句；(3) 不能出现取内联函数地址之类的指令；(4) 不能将成员函数定义为虚函数。

内联使函数的作用域局限于当前程序文件，因此全局的即非 static 的 main 函数是不能内联的。另一方面，操作系统必须通过函数调用调用 main 函数。

三．习题答案

2.1 若给出声明：

```
char    c, *pc;
const   char    cc='a';
const   char    *pcc;
char    *const   cpc=&c;
const   char * const cpcc=&cc;
char    *const *pcpc;
```

则下面的赋值哪些是合法的？哪些是非法的？为什么？

(1) c=cc;　　(10) *pc="ABCD"[2];
(2) cc=c;　　(11) cc='a';
(3) pcc=&c;　　(12) *cpc=*pc;
(4) pcc=&cc;　　(13) pc=*pcpc;
(5) pc=&c;　　(14) **pcpc=*pc;
(6) pc=&cc,　　(15) *pc=**pcpc;
(7) pc=pcc;　　(16) *pcc='b';
(8) pc=cpcc;　　(17) *pcpc='c';
(9) cpc=pc;　　(18) *cpcc='d';

解：(1) 合法，c 不是 const 类型的变量，可以赋值。

(2) 非法，cc 是 const 类型的变量，不能赋值。

(3) 合法，pcc 不是 const 类型的指针变量，可以指向非 const 类型的字符变量。

(4) 合法，pcc 不是 const 类型的变量，赋值后指向的 cc 的类型为 const char，同其要求指向的类型一致。

(5) 合法，pc 不是 const 类型的指针变量，赋值后指向的 c 的类型为 char，同其要求指向的类型一致。

(6) 非法，pc 要求指向 char 类型的变量，不能用 const char*类型的&c 赋值。
(7) 非法，pc 要求指向 char 类型的变量，不能用指向 const char*类型的 pcc 赋值。
(8) 非法，pc 要求指向 char 类型的变量，不能用指向 const char*类型的 cpcc 赋值。
(9) 非法，cpc 是 const 类型的变量，不能赋值。
(10) 合法，pc 指向的是非 const 类型的变量，可以赋值，等价于*pc='C'。
(11) 非法，cc 是 const 类型的变量，不能赋值。
(12) 合法，cpc 指向的是非 const 类型的变量，可以赋值。
(13) 合法，pc 是非 const 类型的指针变量，可以用 char *类型的值*pcpc 赋值。
(14) 合法，**pcpc 代表的是非 const 类型的字符变量，可以任何字符类型的值赋值。
(15) 合法，*pc 代表的是非 const 类型的字符变量，可以任何字符类型的值赋值。
(16) 非法，*pcc 代表的字符是 const 类型的字符变量，不能赋值。
(17) 非法，*pcpc 代表的是 const 类型的指针变量，不能赋值。
(18) 非法，*cpcc 代表的是 const 类型的只读变量，不能赋值。

2.2 头文件 string.h 定义了函数原型 char *strcat(char *dest, const char *src)。定义函数 strcat 的函数体，使其将 src 指示的字符串添加到 dest 指示的字符串的后面，并将调用时 dest 的值作为函数的返回值。

解：注意 strcat 的返回值和传入第一个参数的值相同。

```
char *strcat(char*dest, const char *src){
    char *temp=dest;
    while(*temp) temp++;
    while(*(temp++)=*(src++));
    return dest;
}
```

2.3 C 按优先级和结合性解释类型，下述声明是什么意思?

(1) typedef void VF_PC_RI(char*, int &)
(2) typedef VF_PC_RI* P_VF_PC_RI;
(3) typedef int &RIFFII(int ,int);
(4) extern VF_PC_RI funca;
(5) extern P_VF_PC_RI ptra;
(6) extern void func1 (P_VF_PC_RI *);
(7) extern P_VF_PC_RI func2 (int c);
(8) P_VF_PC_RI func3 (P_VF_PC_RI a);
(9) typedef void (*(**VF_PA_P_PF_V(void))[ ]) (const int);

解：(1) 定义一个名为 VF_PC_RI 的类型，该类型定义了一个参数为(char*, int &)没有返回值的函数。
(2) 定义一个名为 P_VF_PC_RI 的类型，该类型定义了一个指向 VF_PC_RI 类型的指针。
(3) 定义一个名为 RIFFII 的类型，该类型定义了一个参数为(int ,int)返回一个引用的函数，该引用引用一个整型量。
(4) 说明一个类型为 VF_PC_RI 的函数 funca。

(5) 说明一个类型为 P_VF_PC_RI 的指针变量 ptra。

(6) 说明一个没有返回值的函数 func1，该函数的参数是一个指向 P_VF_PC_RI 类型的指针。

(7) 说明一个参数为 int 类型的函数 func2，该函数的返回值是一个 P_VF_PC_RI 类型的指针。

(8) 说明一个参数为 P_VF_PC_RI 类型的函数 func3，该函数的返回值是一个 P_VF_PC_RI 类型的指针。

(9) 定义一个名为 VF_PA_P_PF_V 的类型，该类型定义了一个没有参数的函数，该函数返回一个指针，该指针又指向另一个指针，被指向的指针指向一个数组，数组的每个元素存放一个函数指针，该函数指针指向一个参数为 const int 类型没有返回值的函数。

2.4 运行如下程序，打印结果 ri 和 gi 相等吗？为什么？删除 printf 语句中的 8，结果有什么变化？再删除 printf 语句中的 7，结果又有什么变化？

```
#include <stdio.h>
int &f( ) {int   i=10;   int   &j=i;   return   j; }
int g ( ) {int   j=20;   return   j; }
void main (void) {
      int   &ri=f( );
      int   rj=g( );
      printf("ri=%d\trj=%d\n", ri, rj, 1, 2, 3, 4, 5, 6, 7, 8);
      int   &gi=f( );
      int   gj=g( );
      printf ("gi=%d\tgj=%d\n", gi, gj);
}
```

解：(1) 打印结果 ri 和 gi 不等。

(2) 这是因为主调函数 main 的变量引用了被调函数 f( )的局部自动变量 i，该变量的内存是位于栈上的某个固定位置，该位置的值会被其他函数调用传递参数改变，例如被调用 rj=g( )和调用 printf("ri=%d\trj=%d\n", ri, rj, 1, 2, 3, 4, 5, 6, 7, 8)改变，因为值参传递也是通过栈完成的。

(3) 删除 printf 语句中的 8，ri 的输出结果由原来的 6 变为 5。

(4) 再删除 printf 语句中的 7，ri 的输出结果由原来的 5 变为 4，即 ri 的值总是打印其值的 printf 函数调用的倒数第三个参数。

2.5 如下声明和定义是否会导致编译错误？

```
float   g(int);
int     g(int);
int     g(int, int y=3);
int     g(int, …);
int     i=g(8);
```

解：(1) 不能定义返回类型仅和 float g(int)不同的函数 int g(int) 。

(2) g(8)在调用时出现二义性，无法确定是调用 int g(int, int y=3)还是 int g(int, …)。

2.6 定义函数求 1 个以上的整数中的最大值 int max(int c, …)，整数个数由参数 c 指定。

解：由于参数个数没有固定，因此要应用省略参数。

```
int max(int c,  ...){
    int m, k, *p=&c+1;
    m=p[0];
    for(k=1; k<c; k++) if(m<p[k]) m=p[k];
    return m;
}
void main( ){
    int m;
    m=max(3, 4, 8, 10);
    m=max(4, 6, 8, 5, 4);
}
```

第 3 章　C++的类

一．基本内容

构造函数、析构函数、显式调用、隐式调用、访问权限、数据成员、成员函数、类型成员、私有成员、保护成员、公有成员、隐含参数，对象初始化，缺省构造函数、缺省构造函数、内联成员函数、内存管理运算符、紧凑方式、松散方式。

二．学习要点

C++的类可使用保留字 class、struct 和 union 定义，在类中可以定义数据成员、成员函数和类型成员，这些成员可以是私有、保护、公有等不同访问权限的成员。class 缺省定义 private 成员即私有成员；struct 和 union 缺省定义 public 成员即公有成员，这种规定保持了同 C 的 struct 和 union 的兼容。如果不想使用缺省的访问权限，可通过 private、protected 和 public 重新设定。

数据成员用于存放对象属性的值，成员函数用于描述对象的操作，类型成员用于定义类的数据成员和成员函数。和类同名的成员函数称为构造函数，和类同名且带有波浪线的成员函数称为析构函数。

构造函数可以有不同类型的参数，因而构造函数是可以重载的。析构函数只有一个隐含参数 this，就析构函数而言这个参数的类型是固定的，因而析构函数是不能进行重载的。构造函数只能在定义变量时被编译程序自动调用，即只能被隐式调用。析构函数除了在变量退出其作用域前被编译程序自动调用外，还可以被程序员直接调用即显式调用。

abort 退出程序时不执行收工函数，因而也就不会析构全局对象。加上 abort 不析构局部对象，因此，全局和局部对象申请的资源都不会被释放。exit 退出时会执行收工函数，使全局对象得到析构，但局部对象不会被析构。由此可见，在使用 abort 和 exit 退出程序之前，最好直接调用析构函数释放资源。但当函数调用的嵌套层次很深时，无法直接调用析构函数释放所有局部对象的资源。在这种情况下，最好使用异常处理机制。

private 成员只能被其所属类的成员函数和友元函数访问，不能被派生类的成员函数访问，除非派生类成员函数为基类的友元函数；protected 成员除可被其所属类的成员函数和友元函数访问外，还可被其所属类的派生类的成员函数访问；public 成员可以被任何成员函数和非成员函数访问。

非静态成员函数都有一个隐含参数 this，this 所指向的对象即调用当前成员函数的对象。如果类没有定义任何构造函数，编译程序就会提供缺省的构造函数，同理编译程序也会提供缺省的析构函数。缺省的构造函数和析构函数只有一个隐含的参数 this。

构造函数和析构函数的 this 参数的类型是固定的，而其他非静态成员函数的 this 参数的类型是可变的。例如，可以出现类似 int A::f( )、int A::f( )const 和 int A::f( )volatile 的非静态成员函数，这些重载函数的 this 参数的类型分别为 A*const this、const A*const this、volatile A*const this。

当类定义了构造函数，或类包含私有成员时，不能采用 C 语言用{…}的形式对对象初始化，而必须使用程序员自定义的或编译程序自动提供的构造函数初始化对象。对用 new 产生的对象数组的初始化，则必须使用只有 this 参数的构造函数。

此外，编译程序还会提供一个缺省的赋值运算重载函数。该成员函数的原型是固定的，例如，类 A 的缺省赋值运算重载函数的原型为 A& operator =(const A&)，该函数的 this 参数的类型同样为 A *const this。

当成员函数在类的体内定义时，该成员函数自动成为内联的成员函数。当然，也可以用 inline 显式地定义内联成员函数，类的所有成员函数都可以定义为内联函数。内联是否成功同第二章普通函数内联的前提条件类似，此外定义虚函数也会导致内联失败。

C 和 C++可以用 malloc 以及 free 等管理内存。函数 malloc 负责分配内存，在分配内存之后，不调用构造函数构造对象；函数 free 负责释放空间，在释放内存之前，不调用析构函数。此外，C++还能用 new 以及 delete 等运算符函数管理内存。运算符 new 负责分配内存，在分配内存之后，调用构造函数构造对象；运算符 delete 负责释放空间，在释放内存之前，调用析构函数析构对象。由于简单类型没有构造函数和析构函数，因此使用 new->delete 和使用 malloc->free 没有区别，甚至两种可以混杂着使用。

简单类型也可以被看作简单的类，至此，面向对象的思想在所有类型上得到统一。例如，int 类型可以用用构造函数的形式初始化，可以以此类形式 int x(3)定义并初始化变量 x=3。同理，对于复杂的类，当其构造函数除 this 外只有一个参数时，可采用简单类型的初始化形式初始化。

类的存储空间既同编译程序有关，也同计算机的字长有关。在不同的系统环境下编译时，相同的类所占用的内存大小可能不同。即使在相同的系统环境下编译，若编译程序采用的对齐方式不同，则同样的类所占用的内存大小也可能不一样。

编译程序采用的对齐方式有两种：紧凑方式和松散方式。如果采用紧凑方式编译，则数据成员之间不会留下任何空闲字节，这种方式编译的程序所占用的内存较少，但访问数据成员时需要的时间较长。如果采用松散方式编译，则程序执行时需要的空间较多，而程序的执行时间较短。

在松散方式下，编译程序根据数据成员的类型进行对齐。如果数据成员的类型为字符等单字节类型，则数据成员可以存放在开始地址能被 1 整除的内存单元；如果数据成员的类型为整型等双字节类型，则数据成员可以存放在开始地址能被 2 整除的内存单元；如果数据成员的类型为长整型等四字节类型，则数据成员可以存放在开始地址能被 4 整除的内存单元；以此类推。

三．习题答案

3.1 集合类的头文件 Set.h 如下，请定义其中的函数成员。

```
class   SET{
    int     *set;                   //set 用于存放集合元素
    int     card;                   //card 为能够存放的元素个数
    int     used;                   //used 为已经存放的元素个数
public:
```

```
        SET(int    card);              //card 为能够存放的元素个数
        ~SET( );
        int    size( );                //返回集合已经存放的元素个数
        int    insert(int v);          //插入 v 成功时返回 1，否则返回 0
        int    remove(int v);          //删除 v 成功时返回 1，否则返回 0
        int    has(int v);             //元素 v 存在时返回 1，否则返回 0
    };
```

解：省略类型说明后，程序如下：

```
    //在此引用上述类型说明
    SET::SET(int    card){
        if (set=new int[card]) SET::card=card;
        used=0;
    }
    SET::~SET( ){
        if(set) { delete set; set=0; card=used=0; }
    }
    int SET::size( ){
        return used;
    }
    int SET::insert(int v){
        if(used<card) {set[used++]=v; return 1;}
        return 0;
    }
    int SET::remove(int v){
        int x;
        if(used>0) {
            for(x=0; x<used; x++)
                if(set[x]==v){
                    used--;
                    for(; x<used; x++)    set[x]=set[x+1];
                    return 1;
                }
            return 0;
        }
        return 0;
    }
    int SET::has(int v){
        int x;
        for(x=0; x<used; x++) if(set[x]==v) return 1;
```

```
    return 0;
}
```

3.2 二叉树类的头文件 node.h 如下，请定义其中的函数成员。

```
class   NODE{
    char    *data;
    NODE    *left, *right;
public:
    NODE(char *);
    NODE(char *data，NODE    *left, NODE *right);
    ~NODE( );
};
```

解：以下程序构造函数申请了内存，必须在析购函数释放。

```
#include <string.h>
//在此引用上述类型说明
NODE::NODE(char *){
    if(NODE::data=new char[strlen(data)+1]){
        strcpy(NODE::data, data);
        left=right=0;
    }
}
NODE::NODE(char *data, NODE    *left, NODE *right){
    if(NODE::data=new char[strlen(data)+1]){
        strcpy(NODE::data, data);
        NODE::left=left;
        NODE::right=right;
    }
}
NODE::~NODE( ){
    if(left) left->~NODE( );
    if(right) right->~NODE( );
    if(data) {delete data; data=0;}
}
```

3.3 队列(queue)就是一种先进先出表。队列通常有插入、删除、测空和清除四种操作。“插入”即将一个元素插到队列尾部；“删除”就是从队列首部取走一个元素；“测空”就是要检查队列是否为空，当队列为空时返回 1，否则返回 0；“清除”就是将队列清空。用链表定义一个字符队列，并定义完成上述操作的公有成员函数。

解：以下程序没有使用循环对列，其出入队列操作次数有限。

```
class QUEUE{
    char *queue;
```

```
    int size, front, rear;
public:
    int insert(char elem);
    int remove(char &elem);
    QUEUE(int size);
    ~QUEUE(void);
};
QUEUE::QUEUE(int sz)
{
    queue=new char[size=sz];
    front=0
    rear=0;
}
QUEUE::~QUEUE(void)
{
    if(queue){
        delete queue;
        queue=0;
        front=0;
        rear=0;
    }
}
int QUEUE::insert(char elem)
{
    if(rear==size) return 0;
    queue[rear++]=elem;
    return 1;
}
int QUEUE::remove(char &elem)
{
    if(front==rear) return 0;
    elem=queue[front=front+1];
    return 1;
}
void main(void){ QUEUE queue(20); }
```

3.4 改用数组定义习题 3.3 中的字符队列，并将该队列定义为循环队列，使队列的“插入”、“删除”操作能并发进行，其他公有函数成员的原型不变。

解：注意，函数成员 insert 和 remove 的尾部带有 volatile，表示函数的隐含参数 this 的类型为 volatile * const this，即 this 指向的当前对象是挥发易变的。挥发性通常可以用来修

饰并行操作的对象，即当前队列是可以同时进行插入和删除操作的。构造函数和析构函数不能用 volatile，因为构造和析构时对象必须是稳定的。以下程序使用字符数组作为循环队列。

```
class QUEUE{
    char *queue;
    int size, front, rear;
public:
    int insert(char elem) volatile;
    int remove(char &elem) volatile;
    QUEUE(int size);
    ~QUEUE(void);
};
QUEUE::QUEUE(int sz)
{
    queue=new char[size=sz];
    front=rear=0;
}
QUEUE::~QUEUE(void)
{
    if(queue){
        delete queue;
        queue=0;
        front=0;
        rear=0;
    }
}
int QUEUE::insert(char elem)volatile
{
    if((rear+1)%size==front) return    0;
    queue[rear=(rear+1)%size]=elem;
    return 1;
}
int QUEUE::remove(char &elem)volatile
{
    if(rear==front) return    0;
    elem=queue[front=(front+1)%size];
    return 1;
}
void main(void)
```

```
{
    QUEUE queue(20);
}
```

3.5 利用 C 的文件类型 FILE，定义新的文件类 DOCU。DOCU 用构造函数打开文件，用析构函数关闭文件，并提供同 fread、fwrite、ftell、fseek 类似的函数成员 read、write、tell、seek。类 DOCU 的声明如下，请定义其中的函数成员。

```
class DOCU{
    char *name;
    FILE *file;
public:
    int read(void *ptr, int size, int n);
    int seek(long offset, int whence);
    int write(const void *ptr, int size, int n);
    long tell( );
    DOCU(const char *filename, const    char *mode);
    ~DOCU( );
};
```

解：构造函数申请了内存及文件句柄资源，必须在析构函数释放。注意，文件句柄不能反复释放。

```
#include <stdio.h>
#include <string.h>
//在此引用上述类型说明
DOCU::DOCU(const char *filename, const    char *mode){
    name=new char[strlen(filename)+1];
    strcpy(name, filename);
    file=fopen(name, mode);
}
int DOCU::read(void *ptr, int size, int n){
    return fread(ptr, size, n, file);
};
int DOCU::write(const void *ptr, int size, int n){
    return fwrite(ptr, size, n, file);
}
int DOCU::seek(long offset, int whence){
    return fseek(file, offset, whence);
}
long DOCU::tell( ){
    return ftell(file);
}
```

3.6 线性表通常提供元素查找、插入和删除等功能。以下线性表是一个整型线性表，表元素存放在动态申请的内存中，请编程定义整型线性表的函数成员。

```
class INTLIST{
    int   *list;              /动态申请的内存的指针
    int   size;               //线性表能够存放的元素个数
    int   used;               //线性表已经存放的元素个数
public:
    INTLIST(int   s);         //s 为线性表能够存放的元素个数
    int insert(int   v);      //插入元素 v 成功时返回 1，否则返回 0
    int remove(int   v);      //删除元素 v 成功时返回 1，否则返回 0
    int find(int   v);        //查找元素 v 成功时返回 1，否则返回 0
    int get(int   k);         //取表的第 k 个元素的值作为返回值
    ~INTLIST(void);
};
```

解：构造函数申请了内存资源，必须在析构函数释放。

```
//在此引用上述类型说明
INTLIST::INTLIST(int s){
    if(list=new int[s]){
        size=s;
        used=0;
    }
}
INTLIST::~INTLIST(void){
    if(list){ delete list; list=0; size=used=0; }
}
int INTLIST::insert(int   v){
    if(used<size){
        list[used++]=v;
        return 1;
    }
    return 0;
}
int INTLIST::remove(int   v){
    for(int i=0; i<used; i++)
        if(list[i]==v) {
            used--;
            for (; i<used; i++) list[i]=list[i+1];
            return 1;
        }
```

```
        return 0;
}
int INTLIST::find(int    v){
        for(int i=0; i<used; i++)
                if(list[i]==v) return 1;
        return 0;
}
```

第 4 章　作用域及成员指针

一．基本内容

单目运算符、双目运算符、作用域运算符、词法单位、作用域、面向过程的作用域、面向对象的作用域、名字空间、成员指针、mutable。

二．学习要点

C++的大部分运算符为单目运算符和双目运算符。单目运算符是指只有一个操作数的运算符，同理，双目运算符是指有两个操作数的运算符。作用域运算符::既是单目运算符，又是双目运算符。作用域运算符的优先级为最高级即第十六级，其结合性是按照自左向右的顺序结合。

单目::用于限定全局的类型名、变量名以及常量名等，双目::用于限定类的枚举元素、数据成员、成员函数以及类型成员等。此外，双目运算符::还用于限定名字空间成员，以及恢复自基类继承的成员的访问权限。在类的体外定义数据和成员函数时，必须用双目::限定类的数据和成员函数。

一个词法单位是指编译程序能够识别的一个单词，一个标识符、一个运算符、一个分隔符或一个常量均是一个词法单位。一个标识符通常又可分为常量标识符、变量标识符、函数标识符和类型标识符。

一个词法单位的作用域是指该词法单位能够被访问的范围。访问是指取值、赋值、调用、取地址等操作。

根据作用域是否同类相关，C++的作用域可以分为两类：面向过程的作用域和面向对象的作用域。面向过程的作用域即 C 传统的作用域，词法单位的作用范围从小到大可以分为四级：①作用于表达式内，②作用于函数内，③作用于程序文件内，④作用于整个程序。具有第 3 种和第 4 种作用域的标识符可用单目::限定。

在面向对象的作用域内，词法单位的作用范围从小到大可以分为五级：①作用于表达式内，②作用于成员函数内，③作用于类或派生类内，④作用于基类内，⑤作用于虚基类内。常量对象的作用域属于第 1 级，即常量对象在表达式内构造和析构；标识符和匿名变量的作用域属于第 2 级到第 5 级。

标识符的作用域越小，访问的优先级别就越高。在成员函数内，当函数参数、局部变量同类的数据成员名称相同时，优先访问的是成员函数的函数参数和局部变量，如果此时希望访问同名的数据成员，就必须用类名和作用域运算符加以限定。

单目运算符::可以限定存储类为 static 和 extern 的全局变量、函数、类型以及枚举元素等。当成员函数调用同其同名的全局函数时，必须用单目::限定同名函数为全局函数。

名字空间是 C++引入的一种新的作用域，用于减少软件项目中的命名冲突。同一名字空间中的标识符名必须唯一，不同名字空间中的标识符名可以相同。当一个程序引用多个名字空间而出现成员同名时，可以用名字空间加作用域运算符限定被引用的成员。

保留字 namespace 用于定义名字空间。名字空间必须在程序的全局作用域内定义，不能在函数及成员函数内定义。名字空间可以嵌套定义，最外层名字空间的名称必须在程序的全局作用域唯一。

名字空间也可分多次定义，即可先在初始定义中定义一部分成员，然后在扩展定义中再定义另一部分成员，或者定义初始定义中声明的函数原型。初始定义和扩展定义的语法格式相同。

没有名称的名字空间称为匿名名字空间，每个程序只能有一个匿名名字空间。许多 C++程序没有定义名字空间，其实这些程序使用的是匿名名字空间。

保留字using用于声明程序要引用名字空间的成员，或者用于指示程序要引用名字空间。在引用名字空间的某个成员之前，该成员必须已经在名字空间中作了声明或进行了定义。

访问名字空间的成员有三种方式：①直接访问成员，②声明引用成员，③指定名字空间。直接访问成员的形式为：<名字空间名称>::<成员名称>，直接访问总能唯一地访问名字空间成员。声明引用成员的形式为 using<名字空间名称>::<成员名称>，以后就可以单独使用成员名称进行访问。指定名字空间的形式为 using namespace <名字空间名称>，之后就可以单独使用名字空间的成员名称进行访问。

在一个名字空间里或在匿名名字空间里，可以使用上述三种方式访问另一个名字空间的成员。如果名字空间 A 包含 using namespace B，则在名字空间 C 指定名字空间 A 后，名字空间 C 的程序就可以访问名字空间 A 及 B 的所有成员。

在声明引用成员和指定名字空间定后，不同名字空间的成员名称可能相同，在这种情况下，必须通过直接访问成员的方式访问同名成员。

在声明引用成员时，被引用的名字空间成员加入当前作用域，因此，不能在当前作用域定义和被引用的名字空间成员同名的成员；在指定名字空间时，不会将任何标识符加入当前作用域，因此，可在当前作用域定义和名字空间中的标识符同名的标识符。当名字空间的成员和匿名名字空间的全局标识符同名时，可以通过单目::指定访问匿名名字空间的全局标识符；当名字空间的成员和匿名名字空间的局部标识符同名时，首先访问的是匿名名字空间的的局部标识符。

可以为名字空间定义别名以代替过长和难懂的名字空间名称，定义形式为 namespace <别名>=<名字空间名称>**::**<子名字空间名称>**::**<子名字空间名称>**::…::** <子名字空间名称>。对于嵌套定义的名字空间，使用别名可以大大提高程序的可读性。

成员指针有两种：普通成员指针和静态成员指针。一般在提及成员指针时主要是指普通成员指针。普通成员指针是形式为<类名>::*的指针，该类指针指向类的普通数据成员和普通函数成员。静态成员指针的形式和 C 的普通指针的形式相同，该类指针指向类的静态数据成员和静态函数成员。

运算符.*和->*均为双目运算符，用于访问普通成员指针所指向的普通成员，它们运算的优先级均为第 14 级，优先级相同时按自左向右的顺序结合。运算符.*的左操作数为类的对象，右操作数为指向成员的指针；运算符->*的左操作数为指向对象的指针，右操作数为指向成员的指针。

当成员指针 P 指向某个数据成员时，不能移动指针将 P 指向其他数据成员，因为数据成员的大小及类型不一定相同；也不能将 P 强制转换为其他任何类型或反之，否则便可以

通过类型转换和加减法运算间接实现指针移动。

成员指针不能移动的原因在于：①移动后指向的内存单元可能属于某个成员的一部分，或者跨越两个或两个以上成员的内存空间；②即使移动后正好指向另一个成员，成员的类型不一定正好相同，个成员的访问权限也不一定相同。此时，通过指针访问成员便可能出现越权访问。

成员函数的参数表后出现 const 表示 this 指向的对象不能修改，确切地讲，即不能修改 this 指向的对象的非静态数据成员，但可以修改 this 指向的对象的非只读类型的静态数据成员。关于静态数据成员和静态成员指针，请参考第五章。

mutable 仅能用于修饰类的非静态数据成员，表示该数据成员为机动数据成员，机动数据成员不能用 const、volatile 以及 static 修饰。当普通成员函数的参数表后出现 const 时，该函数不能修改对象的任何非静态数据成员。但是，如果将非静态数据成员的存储类说明为 mutable，则该数据成员就可以被参数表后出现 const 的成员函数修改。

任何标识符都可以定义为引用。被引用的对象可以是简单类型的变量，也可以是复杂类型的变量或对象。引用变量是被引用对象的别名，被引用的对象必须进行构造和析构，而引用变量本身不必要进行构造和析构。但是，如果 A 类引用变量 r 引用了 new 生成的对象，即通过类似 A &r=*new A( )引用的对象，那么就应该使用 delete　&r 析构该对象。

左值引用即非 const 类型的引用，左值引用变量即非 const 类型的引用变量，左值引用参数即非 const 类型的引用参数。左值引用变量必须用同类型的左值表达式初始化，左值引用参数必须用同类型的左值表达式传递实参。如果初始化表达式的类型不符，就会产生类型相符的匿名变量以供引用。

左值引用参数相当于换名形参，可将函数对实参的修改结果带出函数体外。但如果产生了匿名变量，则修改结果就存放在匿名变量中，而匿名变量在调用结束后是不可访问的，此时的左值引用参数就不能起到换名形参的作用。

非引用类型的形参相当于作用域局限于函数的局部变量，因此，形参所存储的对象的析构是在函数调用返回前即作用域结束前完成的。形参所存储的对象的构造则是在调用函数时通过值参传递完成的。值参传递将实参各数据成员的值对应地赋给形参的数据成员，对于指针类型的数据成员只复制指针的值，而没有复制指针所指向的存储单元的内容。

值参传递所进行的赋值又称为浅拷贝赋值，浅拷贝赋值导致形参和实参的指针数据成员指向共同的存储单元。由此而造成的后果是：一旦调用返回导致形参所存储的对象析构，就必然会释放其指针数据成员所指向的存储单元，释放的存储单元可能立即被操作系统分配给其他程序，但程序本身并不知道该存储单元已分配给其他程序，若它通过调用时使用的实参变量继续访问该存储单元，就会造成一个程序非法访问另一个程序的内存单元，从而导致操作系统产生页面保护错误并终止程序。

必须定义深拷贝构造函数才能避免出现页面保护错误。类 A 的深拷贝构造函数的原型最好定义为 A(const A&)。

三、习题答案

4.1 为什么要引入名字空间？名字空间能否在函数内部定义？

解：名字空间是 C++引入的一种新的作用域，用于减少软件项目中的命名冲突。同一名字

空间中的标识符名必须唯一，不同名字空间中的标识符名可以相同。

名字空间必须在程序的全局作用域内定义，不能在函数及函数成员内定义。

4.2 匿名名字空间能否分多次定义？它和没有对象的匿名联合有何区别？

解：名字空间包括匿名名字空间可以分多次定义，即可以先在初始定义中定义一部分成员，然后在扩展定义中再定义另一部分成员，或者再定义初始定义声明的函数原型。

匿名名字空间的作用域规则和没有对象的匿名联合相同。但匿名名字空间不能看作类，尽管它可以定义类型、变量及函数等。匿名名字空间的局部变量内存是独立分配的，而没有对象的匿名联合的成员变量是共享内存的。匿名名字空间只能在函数的外面定义，而没有对象的匿名联合可在函数的内部定义。

4.3 为统计正文的单词定义一个类，其类型声明的头文件 word.h 如下：

```
class WORD{
      char    *word;
      int      count;
public:
      int gettimes( )const;
      int inctimes( );
      const char *getword( )const;
      WORD(const char *);
      ~WORD( );
};
```

定义其中的函数成员，要求构造函数使用 new 为结点分配空间，析构函数使用 delete 回收分配的空间。

解：构造函数申请了内存资源，必须在析构函数释放。

```
//在此引用上述类型说明
   int WORD::gettimes( )const{ return count; }
   int WORD::inctimes( ){ return ++count; }
   const char *WORD::getword( )const{ return word; }
   WORD::WORD(const char *w){
        if (word=new char[strlen(w)+1]) strcpy(word, w);
        count=0;
   }
   WORD::~WORD( ){ if (word) {delete word; word=0; }}
```

4.4 利用上述 WORD 类实现一个单词表，其类型 WORDS 定义如下：

```
#include <string.h>
#include <iostream.h>
#include "word.h"
class WORDS{
      WORD *words;
      int    count, total;
```

```
public:
    int insert(const char *);
    WORD *const find(const char *);
    WORDS(int total);
    ~WORDS( );
};
void main(void)
{
    WORDS   ws(20);
    ws.insert("amour");
    ws.find("amour")->gettimes( );
    ws.find("amour")->inctimes( );
    cout<<"Times of amour=";
    cout<<ws.find("amour")->gettimes( );
}
```

请定义其中的函数成员，并用 main 进行测试。

解：构造函数申请了内存资源，必须在析构函数释放。尤其要注意 new (&words[count++]) WORD(w)的用法，表示利用已有对象的存储单元重新构造该对象。WORDS 的函数成员定义如下：

```
#include <string.h>
#include <alloc.h>
WORDS::WORDS(int total){
    words=(WORD *) malloc(total*sizeof(WORD));
    if (words) { count=0; WORDS::total=total; }
}
WORDS::~WORDS( ){
    for(int x=0; x<count; x++) words[x].~WORD( );
    free(words);
}
int WORDS::insert(const char *w){
    if(find(w)) return 0;
    new (&words[count++]) WORD(w);
    return 1;
}
WORD *const WORDS::find(const char *w){
    for (int k=0; k<count; k++)
     if(strcmp(w, words[k].getword( ))==0)return &words[k];
    return 0;
}
```

4.5 字符串类的类型声明如下：

```
#include <string.h>
#include <iostream.h>
class STRING{
    char *str;
public:
    int strlen( )const;
    int strcmp(const STRING &)const;
    STRING &strcpy(const STRING &);
    STRING &strcat(const STRING &);
    STRING(char *);
    ~STRING( );
};
void main(void)
{
    STRING s1("I like apple");
    STRING s2(" and    pear");
    STRING s3(" and    orange");
    cout<<"Length of s1="<<s1.strlen( )<<"\n";
    s1.strcat(s2).strcat(s3);
    cout<<"Length of s1="<<s1.strlen( )<<"\n";
    s3.strcpy(s2).strcpy(s1);
    cout<<"Length of s3="<<s3.strlen( )<<"\n";
}
```

试定义字符串复制及连接等函数成员，这些函数成员调用 C 的字符串运算函数。

解：构造函数申请了内存资源，必须在析构函数释放。注意定义 STRING &strcpy 和 STRING &strcat，此类定义容许 strcpy 和 strcat 连续运算，参见 main 函数中的连续运算。STRING 的函数成员定义如下：

```
int STRING::strlen( )const{ return ::strlen(str); }
int STRING::strcmp(const STRING &s)const{return ::strcmp(str, s.str); }
STRING &STRING::strcat(const STRING &s){
    int    len=::strlen(str)+::strlen(s.str)+1;
    char *t=str;
    if(str=new char[len]) {
        ::strcat(::strcpy(str, t), s.str);
    }
    delete t;
    return *this;
}
```

```
STRING &STRING::strcpy(const STRING &s){
    int    len=::strlen(s.str)+1;
    delete str;
    if(str=new char[len]) ::strcpy(str, s.str);
    return *this;
}
STRING::STRING(char *s){
    if(str=new char[::strlen(s)+1])    ::strcpy(str, s);
}
STRING::~STRING( ){ if (str) { delete str;    str=0; } }
```

4.6 定义一个杂凑表类。要求用数组存放杂凑表元素，以字符串作关键字存放和查找表元素，并提供用于插入、查询和删除杂凑表元素的 public 函数成员。

```
#include <string.h>
#include <alloc.h>
class HASH{
    class NODE{
        int value;
        NODE *next;
    public:
        int getvalue( ){ return value;}
        NODE *getnext( ){ return next;}
        NODE *setnext(NODE *n){ next=n; return this;}
        NODE(int v, NODE *n){ value=v; next=n; }
        ~NODE( ){ if(next) delete next; }
    };
    struct BARREL{ char *s; NODE *n;}*h;
    int    c;
    const int    t;
public:
    HASH(int m);
    NODE *lookup(const char *s, int v);
    int insert(const char *s, int v);
    int remove(const char *s, int v);
    ~HASH( );
};
```

解：注意 NODE 是局部于 HASH 的类型，请注意以下 lookup 函数的返回类型。

```
//在此引用上述类型说明
HASH::HASH(int m):t((h=new BARREL[m])?m:0),c(0){ }
HASH::NODE *HASH::lookup(const char *s, int v){
```

```
    for(int k=0; k<c; k++)
        if(strcmp(s, h[k].s)==0){
            NODE *p=h[k].n;
            while(p!=0){
                if(p->getvalue( )==v) return p;
                p=p->getnext( );
            }
        }
    return 0;
}
int HASH::insert(const char *s, int v){
    for(int k=0; k<c; k++)
        if(strcmp(s, h[k].s)==0){
            h[k].n=new NODE(v, h[k].n);
            return 1;
        }
    if(c<t){
        h[c].s=new char[strlen(s)+1];
        strcpy(h[c].s, s);
        h[c++].n=new NODE(v, 0);
        return 1;
    }
    return 0;
}
int HASH::remove(const char *s, int v){
    for(int k=0; k<c; k++)
        if(strcmp(s, h[k].s)==0){
            NODE *p,*q=h[k].n;
            if((p=q)==0) return 0;
            if(p->getvalue( )==v){
                h[k].n=p->getnext( );
                free(p);
                return 1;
            }
            while(p=p->getnext( )){
                if(p->getvalue( )==v) {
                    q->setnext(p->getnext( ));
                    free(p);
                    return 1;
```

```
                    }
                    q=p;
                }
            }
        return 0;
}
HASH::~HASH( ){
    for(int k=0; k<c; k++)    delete h[k].n;
    delete h;
}
void main( ){
    HASH h(10);
    h.insert("A", 1);
    h.insert("A", 2);
    h.remove("A", 1);
    h.insert("A", 3);
    h.insert("B", 1);
    h.insert("B", 2);
}
```

4.7 定义学生成绩记录。记录包含姓名、密码等标示信息，以及英语、数学、物理、化学等成绩信息。在查询某个学生的某科成绩时，要求正确提供该学生的的密码。

解：在以下程序中，密码核对正确后，通过返回相应成员的指针取得某科成绩，这正是成员指针的恰当用法。

```
#include <conio.h>
#include <string.h>
#include <iostream.h>
class RECORD{
    char password[10];
    int    english, mathematics,physics,chemistry;
public:
    char name[10];
    int RECORD::* get(char *item, char * pswd);//密码核对正确后，返回成员指针
    RECORD(char *name, char *pswd, int engl, int math, int phys, int chem);
};
RECORD::RECORD(char *name, char *pswd, int engl, int math, int phys, int chem)
{
    strcpy(RECORD::name, name);
    strcpy(password, pswd);
    english=engl;
```

```
    mathematics=math;
    physics=phys;
    chemistry=chem;
}
int RECORD::* RECORD::get(char *item, char *pswd)
{
    if(stricmp(pswd, password)) return 0;     //在 C++中返回 0 表示空指针
    if(stricmp(item, "english")==0) return &RECORD::english;
    if(stricmp(item, "mathematics")==0) return &RECORD::mathematics;
    if(stricmp(item, "physics")==0) return &RECORD::english;
    if(stricmp(item, "chemistry")==0) return &RECORD::mathematics;
    return 0;                                 // C++提倡返回 0 表示空指针
}
char *getpswd(const char *name)
{
    int   i=0;
    static char pswd[10];
    cout<<"Mr. "<<name<<", please input your password: ";
    while((pswd[i]=getch( ))!='\r') if(i<9) { i++; }
    pswd[i]=0;
    cout<<"\n\n";
    return pswd;
}
RECORD yang("Yang", "123456789", 88, 98, 89, 93);
RECORD wang("Wang", "abcdefghi", 98, 89, 98, 97);
void main(void) {
    char *pswd=getpswd(yang.name);
    int   RECORD::*p;                         //定义数据成员指针 p
    p=y->get("english", pswd);
    if(p==0) {
        cout<<"Password or inquiry item does not exist!\n";
        return;
    }
    cout<<"Your english is "<<yang->*p<<"\n";
}
```

4.8 定义二维坐标系上的三角形类。要求该类提供计算面积和周长的函数成员，计算时不能改变类的任何数据成员的值。

解：由于计算时不能改变类当前对象任何数据成员的值，故计算函数的隐含参数 this 必须用 const 修饰，即在计算函数的参数表后加上 const。

```
#include <math.h>
class TRIANGLE{
    double x1, y1, x2, y2, x3, y3;
public:
    TRIANGLE(double x1, double y1, double x2, double y2, double x3, double y3);
    double area( )const;
    double perimeter( ) const;
};
TRIANGLE::TRIANGLE(double x1,double y1,double x2,double y2,double x3,double y3)
{
    TRIANGLE::x1=x1;
    TRIANGLE::y1=y1;
    TRIANGLE::x2=x2;
    TRIANGLE::y2=y2;
    TRIANGLE::x3=x3;
    TRIANGLE::y3=y3;
}
double TRIANGLE::area( ) const {
    return   abs((y1+y3)*(x3-x1)+(y1+y2)*(x1-x2)+(y2+y3)*(x2-x3))/2;
}
double TRIANGLE::perimeter( ) const{
    double s1= sqrt(pow(x1-x2,2)+pow(y1-y2,2));
    double s2=sqrt(pow(x1-x3,2)+pow(y1-y3,2));
    double s3= sqrt(pow(x3-x2, 2)+pow(y3-y2,2))
    return   s1+s2+s3;
}
```

4.9 指出如下程序中的错误。

```
struct A{
    char   *a, b, *geta( );
    char   A::*p;
    char   *A::*q( );
    char   *(A::*r)( );
};
void main(void) {
    A a;
    a.p=&A::a;
    a.p=&A::b;
    a.q=&A::geta;
    a.r=a.geta;
```

```
}
```

解：错误出现在 main 中，主要是赋值表达式的类型匹配问题。错误见以下 main 中的注解。

```
void main(void) {
    A a;
    a.p=&A::a;      //不能将&A::a 的类型 char *A::*转换为 a.p 的类型 char A::*
    a.p=&A::b;
    a.q=&A::geta;  //a.q 既不是取函数 q 的地址，也不是调用 q 函数
    a.r=a.geta;     //a.geta 既不是取函数 geta 的地址，也不是调用 geta 函数
}
```

4.10完成如下栈类 STACK 的函数成员定义，STACK 类的头文件 stack.h 的内容如下：

```
#include <string.h>
#include <iostream.h>
class   STACK{
    const   int   max;   //栈能存放的最大元素个数
    int      top;          //栈顶元素位置
    char     *stk;
public:
    STACK(int   max);
    ~STACK( );
    int   push(char v);   //将 v 压栈，成功时返回 1，否则返回 0
    int   pop(char&v);   //弹出栈顶元素，成功时返回 1，否则返回 0
};
```

解：注意 STACK 中的 const 数据成员 max，其初始化不能在构造函数的函数体内完成。此外，引用类型和类类型的成员也不能在构造函数的函数体内完成。

```
//在此引用上述类型说明
STACK::STACK(int   max): top(0), max((stk=new char[max])?max:0) {   }
STACK::~STACK( ){ if(stk) { delete stk;   stk=0; } }
int STACK::push(char   v){
    if (top>=max) return 0;
    stk[top++]=v;
    return 1;
}
int STACK::pop(char &v){
    if (top<=1) return 0;
    v=stk[--top]=v;
    return 1;
}
```

第 5 章　静态成员与友元

一．基本内容

静态成员、静态数据成员、静态函数成员、类变量、类实例变量、静态成员指针、友员函数、成员友员、普通友员。

二．学习要点

类可以用 static 定义静态成员，静态成员又分为静态数据成员和静态函数成员。由 class、struct 和 union 定义的类可以定义静态函数成员，但只有由 class 和 struct 定义的类才可以定义静态数据成员。

在访问权限方面，静态成员和普通成员一样。静态成员有三种访问形式：① <类名>::<成员名>；② <对象>.<类名>::<成员名>；③ <对象>.<成员名>。第一种形式表明静态成员可以脱离对象存在，是 C++提倡的静态成员访问形式。实际上，其它两种形式的访问都被自动转换为第一种形式访问。因此，改变任何对象的静态数据成员的值也就改变了其他对象的静态数据成员的值，同时也就改变了类的静态数据成员的值。

静态数据成员用于描述类的全局信息，从面向对象的角度来看，静态数据成员相当于类变量，而普通数据成员相当于类实例变量。类变量用来存储类的全局信息，如类的对象总数、连接所有对象的链表表头等。

由此可知，静态数据成员只与类型相关，可脱离类类实例即对象独立存在，其存储单元不属于任何对象，是一处独立分配的内存单元。因此，静态数据成员同 C 语言的变量没有太大区别。不同的是，静态数据成员有 private、protected、public 等不同的访问权限。鉴于静态数据成员的空间独立性，对于类 A 的对象 a 来说，sizeof(A) 等于 sizeof(a)等于除静态数据成员之外的成员内存之和。

静态数据成员必须在类的体内声明，在类的体外定义即分配内存并初始化。在类体内定义的类称为嵌套类，在函数体内定义的类称为局部类，某些编译程序不允许定义局部类。标准 C++允许定义局部类，但不允许局部类定义静态数据成员，否则会造成静态数据成员的生存矛盾。

静态函数成员的访问权限及继承规则同普通函数成员没有区别，同样，静态函数成员也可以缺省参数、省略参数以及进行重载。不同的是，普通函数成员有一个隐含参数 this，而静态函数成员则没有隐含参数 this。含有 this 参数的成员函数不能用 static 定义，否则，在是否包含 this 参数的问题上就会自相矛盾。

构造函数、析构函数以及虚函数等都是有 this 参数的。此外，若函数成员的参数表后出现了 const 或 volatile，则表明该函数成员是有隐含 this 参数的。因此，这些函数不能定义为静态函数成员。

联合 union 不能定义静态数据成员，但可以定义静态函数成员。局部类不能定义静态数据成员，但是却能够定义静态函数成员。这样的静态函数成员必然为内联函数，因为 C ++

不允许在函数体内嵌套定义函数。

静态数据成员除了具有访问权限外，同普通变量没有本质上的区别，因此，静态数据成员指针和普通变量指针没有形式上的区别。但从另一方面来说，静态数据成员指针和普通数据成员指针就有很大差别。在计算含有指针变量的表达式时，要注意各类指针的优先级和结合性。

将函数声明为友元函数，就像将函数声明为内联函数一样，可以提高这些函数的执行效率。友元函数（简称友元）用 friend 声明，虽然不是声明该友元的类的函数成员，但是，它能像类的函数成员一样访问类的所有成员。友元分普通友元和成员友元两种类型。普通友元是用类似 C 的普通函数声明的某个类的友元，成员友元是用一个类的函数成员声明的另一个类的友元。

由于友元不是声明该友元的类的函数成员，故不受该类的访问权限的限制，因此，在 private、protected 或在 public 下声明没有区别。如果声明友元函数时同时定义了友元函数的函数体，这样的友元函数将自动成为内联函数。

全局 main 函数也可以声明为某个类的普通友元。但声明时不能同时定义其函数体，即不能同时定义为内联函数。因为内联函数的作用域是局限于当前文件的，操作系统也无法调用这样的 main 函数。

构造函数和析构函数也可以声明为另一个类的成员友元，由于构造函数和析构函数没有返回类型，故声明为成员友元时可以随意指定它们的返回类型。C++将这一规定放宽到声明任何成员友元。

一个类的任何成员函数都能声明为另一个类的成员友元，且可以声明为多个类的成员友元。如果一个类的所有成员函数都要成为另一个类的友元，则可以用 friend <类名>的形式作简要声明。例如，在类 B 中声明 friend A、friend class A 或 friend struct A，都是将类 A 的所有成员函数声明为类 B 的成员友元。

三．习题答案

5.1 定义一个类 RANDOM，该类的构造函数用来指定随机数的分布参数，在对象不同而分布参数相同的情况下，构造函数自动地使分布参数互不相同；该类的函数 rand 用于返回下一个随机数。

解：在面向对象的程序设计中，注意应用对象识别标志的唯一性。即使对象的构造函数的参数完全相同，但对象的识别标志是唯一性的，据此可使随机数的分布参数互不相同。在 C++中，对象的首地址即隐含参数 this 可作为对象的唯一性识别标志。

```
class RANDOM{
    int m, r, f, y;
public:
    RANDOM(int mod, int ran, int fac);
    int getran( );
};
RANDOM::RANDOM(int mod, int ran, int fac)
{
```

```
        m=mod;
        r=ran;
        f=fac;
        y=(int) this;
    }
    int RANDOM::getran( ){ return r=(r*f+y)%m; }
```

5.2 定义节点类 NODE 和栈类 STACK，栈元素由节点构成并形成节点链表，压栈时将节点加入栈的链首，出栈时从栈的链首删除节点。

解：注意 NODE 的构造函数，提供了为 STACK 元素形成链表的潜在功能。

```
    class   NODE{
        int       value;
        NODE *next;
    public:
        NODE(int value, NODE *next);
        int getv( ){ return    value; };
        NODE *getn ( ){ return    next; };
    };
    NODE::NODE(int v, NODE* n){    value=v;    next=n;    }
    class   STACK{
        NODE *head;
    public:
        STACK( ){ head=0; };
        int push(v);
        int pop(int &v);
        ~STACK( );
    };
    int STACK::push(int v)
    {
        NODE *p=new NODE(v, head);
        if(p!=0) head=p;
        return p!=0;
    }
    int STACK::pop(int &v )
    {
        NODE *p=head;
        if(head==0) return 0;
        v=head->getv( );
        head=head->getn( );
        delete    p;
```

```
        return 1;
    }
    STACK::~STACK( )
    {
        NODE *q, *p=head;
        while(p!=0){
            q=p->getn( );
            delete  p;
            p=q;
        }
    }
```

5.3 分析如下定义是否正确，并指出错误原因。

```
    struct  A{
        const  static  int x;
        static  const  int y;
        static  const  int z;
    };
    const  int  A::x=5;
    static  const  int  A::y=6;
    static  int  A::z=7;
```

解：static const int A::y=6 中的 static 应去掉，static int A::z=7 中 static 应改为 const。在 C++中，静态数据成员和静态函数成员在类的外面定义时不能使用 static，否则就会产生教材上所说的对象生存期矛盾。

5.4 分析如下定义是否正确，并指出错误原因。

```
    class A{
        static  int  *j, A::*a, i[5];
    public:
        int  x;
        static  int  &k, *n;
    };
    int  y;
    int  A::i[5]={1, 2, 3};
    int  *A::j=&y;
    int  A::*j=&A::x;
    int  A::*A::a=&A::x;
    int  &A::k=y;
    int  *A::n=&y;
```

解：上述定义都是正确的。

5.5 定义三维坐标系的 POINT 类，并完成 POINT 声明中的函数成员定义。

```
class    POINT{
        int        x, y, z;
        static    int    total;          //点的总个数
public:
        POINT( );
        POINT(int, int, int);
        ~POINT( );
        static int number( );          //返回点的个数
};
```

解：静态数据成员 total 用于存放类型的总体信息，在面向的对象的思想中称之为类变量，而普通数据成员如 x 等则称为类实例变量。同理，number 和 POINT( )等分别被称为类函数和类实例函数。类的总体信息即对象个数 total 可通过构造函数和析构维护。

```
//在此引用上述类型说明
int POINT::total=0;
POINT::POINT( ){ x=y=z=0;    total++; }
POINT::POINT(int x, int y, int z){
        POINT::x=x;
        POINT::y=y;
        POINT::z=z;
        total++;
}
POINT::~POINT( ){ total--; }
int POINT::number( ) { return total; }
```

5.6 现有一棵二叉树和一个有序数组，要求通过这棵二叉树构造有序数组。它们的类型声明如下，请定义有序数组的构造函数。

```
class    SEQUENCE;
class    TREE{
        int    item;
        TREE *left, *right;
        friend    SEQUENCE;
public:
        //请在此定义函数
};
class SEQUENCE{
        int    *items;
        int    size;
public:
        SEQUENCE(TREE &);
};
```

解：本习题主要考查如何为对象添加所需要的函数成员。

```
class   SEQUENCE;
class   TREE{
        int   item;
        TREE *left, *right;
        friend   SEQUENCE;
public:
        int   getnodes( );
        int   getnodes(int *items);
        TREE(int v, TREE*l, TREE*r):item(v), left(l), right(r){ };
};
int TREE::getnodes( ){
        int   l=0, r=0;
        if(left) l=left->getnodes( );
        if(right) r=right->getnodes( );
        return l+r+1;
}
int TREE::getnodes(int *items){
        int   n=0;
        if(left) n=left->getnodes(items);
        items[n++]=TREE::item;
        if(right) n=n+right->getnodes(items);
        return n;
}
class SEQUENCE{
        int   *items;
        int   size;
public:
        SEQUENCE(TREE &);
};
SEQUENCE::SEQUENCE(TREE &t){
        int   m;
        items=new int[size=t.getnodes( )];
        t.getnodes(items);
}
```

5.7 现欲通过有序数组构造二叉树，请定义二叉树的构造函数(提示：可将数组平均分成两个部分，将中间的数组元素作为根，将二边的数组元素分别作为根的左、右子树，重复上述过程便可以得到基本平衡的二叉树)。

解：任何函数都是可以递归调用的，但要注意控制递归结束的条件，以下程序通过递归调

用构造函数构造二叉树。

```
class   TREE{
        int    item;
        TREE *left, *right;
public:
        TREE(int *a, int t);
};
TREE::TREE(int *a, int t){
        int x=t/2;
        item=a[x];
        left=(x>1)?new TREE(a, x - 1):0;
        right=(t>x+1)?new TREE(a+x+1, t-x-1):0;
}
void   main( ){
        static   int   s[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        TREE   t(s, 10);
}
```

5.8 请指出如下程序中的错误之处：

```
class A{
        int       a;
        static   friend   int   f( );
        friend   int   g( );
public:
        friend   int   A( );
        A(int);
}a(5);
int g( ) { return   a.a; }
```

解：在 static friend int f( )中，static 和 friend 不能同时出现。注意不要把 int A( )当作构造函数，该非成员函数可以在函数 g 的后面定义为 int A( ){ return 3; }。但是，如果在函数名前加上 A::则表示要定义成员函数，显然，friend int A::A( )是错误的，因为 friend 不能用于修饰当前类的成员函数。而说明 A::A(int)则是正确的，因为 A::A(int)出现在类 A 的定义中，故一般情况下 A::是可以省略的。

5.9 定义类描述有限状态自动机，状态的输入和输出关系可以描述为链表数据成员：

```
class   STATE;
class   LIST{
        LIST   *next;
        char   input;
        STATE *output;
        LIST(char in, STATE *out);                //私有，仅供 STATE 使用
```

```
    ~LIST( );
    friend    STATE;
};
class STATE{
    char *name;                          //状态名
    LIST *list;                          //输入及输出列表
    static STATE*error;
public:
    void enlist(char in, STATE *out);    //插入 list
    const STATE *next(char in)const;     //输入 in 转移到下一个状态
    const STATE *start(char *)const;     //启动有限自动机
    STATE(char *name);
    ~STATE( );
};
```

使用有限自动机编程解决如下问题：有一个人带着狼、羊和草来到河的左岸，左岸只有一条无人摆渡的船。这个人要从左岸过河到右岸，可是这条船最多只能装一个人和其他三者之一，否则便会沉没。如果没有人看管，狼会吃掉羊，或者羊吃掉草。问如何过河才能保证羊和草的安全。

提示：作为有限状态自动机的输入，人单独过河用字符 m 表示，人带狼过河用字符 w 表示，人带羊过河用字符 s 表示，人带草过河用字符 g 表示，声明有限自动机的 start、stop 以及 error 状态对象，如果 start.start("smwsgms")=&stop，则过河成功，否则，如果 start.start("smwsgms") ==&error，则过河失败。

解：以下定义的类 STATE 用于表示自动机的状态，其中 static STATE*error 表示自动机的错误陷阱，进入错误陷阱后其他任何输入都不能改变自动机的状态。开始时，人狼羊草都在河的左岸，初试状态 start 表示为"WSGM_"，人带着羊过河后状态变为"WG_SM"，用 start.enlist('S', &WG_SM)将这一操作加入状态转移表。自动机的状态转移表存放类 LIST 中，由以下自动的状态转移图可知，本题有两个过河路径最小解。

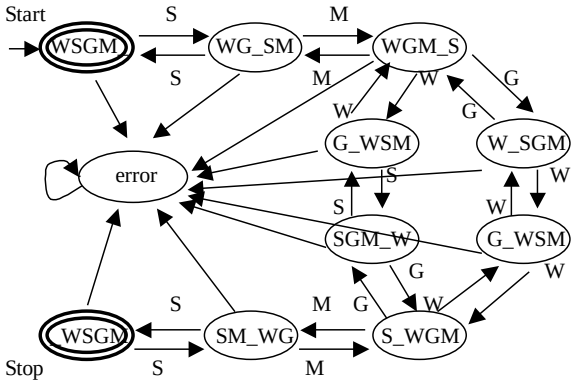

图 5.1　人带狼羊草过河的有限状态自动机

```
#include <string.h>
#include <iostream.h>
class    STATE;
class    LIST{
      LIST    *next;
      char     input;
      STATE *output;
      LIST(char in, STATE *out);          //私有，仅供 STATE 使用
      ~LIST( );
      friend    STATE;
};
LIST::LIST(char in, STATE *out){
      input=in;
      output=out;
      next=0;
}
LIST::~LIST( ){
      if(this->next){ delete this->next;}
}
class STATE{
      char     *name;                     //状态名
      LIST    *list;                      //输入及输出状态转移列表
      static    STATE*error;
public:
      void enlist(char in, STATE *out);    //插入 list
      const STATE *next(char in)const;   //输入 in 转移到下一个状态
      const STATE *start(char *)const;    //启动有限自动机
      STATE(char *name);
      ~STATE( );
};
STATE *STATE::error=0;
STATE::STATE(char *name):name(0),list(0){
      if(name==0){ error=this; return; }
      STATE::name=new char[strlen(name)+1];
      strcpy(STATE::name, name);
}
void STATE::enlist(char in, STATE *out){//插入 list
```

```
    LIST *temp;
    if(list==0){
        list=new LIST(in, out);
        return;
    }
    temp=new LIST(in, out);
    temp->next=list;
    list=temp;
}
const STATE *STATE::next(char in)const{    //输入 in 转移到下一个状态
    LIST *temp=list;
    if(this==error) return error;
    while(temp)
        if(temp->input==in) return temp->output;
        else temp=temp->next;
    return error;
}
const STATE *STATE::start(char *s)const{    //启动有限自动机
    const STATE *temp=this;
    while(*s) temp=temp->next(*s++);
    return temp;
}
STATE::~STATE( ){
    if(name) {cout<<name<<"\n"; delete name; name=0; }
    if(list) {delete list; list=0; }
}
void main( ){
    STATE start("WSGM_");                  //自动机的启动状态
    STATE stop("_WSGM");                   //自动机的停止状态
    STATE error(0);                        //自动机的出错状态
    STATE WG_SM("WG_SM");
    STATE WGM_S("WGM_S");
    STATE G_WSM("G_WSM");
    STATE SGM_W("SGM_W");
    STATE W_SGM("W_SGM");
    STATE WSM_G("WSM_G");
    STATE S_WGM("S_WGM");
    STATE SM_WG("SM_WG");
    start.enlist('S', &WG_SM);
```

```
    WG_SM.enlist('S', &start);
    WG_SM.enlist('M', &WGM_S);
    WGM_S.enlist('M', &WG_SM);
    WGM_S.enlist('W', &G_WSM);
    WGM_S.enlist('G', &W_SGM);
    G_WSM.enlist('W', &WGM_S);
    W_SGM.enlist('G', &WGM_S);
    G_WSM.enlist('S', &SGM_W);
    SGM_W.enlist('S', &G_WSM);
    SGM_W.enlist('G', &S_WGM);
    S_WGM.enlist('G', &SGM_W);
    W_SGM.enlist('S', &WSM_G);
    WSM_G.enlist('S', &W_SGM);
    WSM_G.enlist('W', &S_WGM);
    S_WGM.enlist('W', &WSM_G);
    S_WGM.enlist('M', &SM_WG);
    SM_WG.enlist('M', &S_WGM);
    SM_WG.enlist('S', &stop);
    stop.enlist('S', &SM_WG);
    if(start.start("SMWSGMSSS")==&stop) cout<<"OK";
}
```

5.10有限自动机也可以解决三个修道士与三个野人的过河问题，假定船最多只能载两个修道士或者野人,野人服从修道士的指挥。无论何时，只要野人多于修道士，野人就会吃掉修道士。以有限自动机为基础，编程解决三个修道士与三个野人的过河问题。

解：自动机仍然可以使用上述定义，只要改写主函数即可。用 M 表示人，用 W 表示野人。初始状态为"MMMYYY_"，过河动作可以有五种：人单独过河 s，两个人过河 d，野人单独过河 w，两个野人过河 t，人带着野人过河 b。例如，从初始状态"MMMYYY_"由人带着野人过河后状态为"_MMMYYY"。

第 6 章　单继承类

一．基本内容

单继承、多继承、基类、派生类、构造顺序、析构顺序。

二．学习要点

继承是 C++类型演化的重要机制。在利用已有类型的数据成员和函数成员的基础上，定义新类只需定义原有类型没有的数据成员和函数成员。新类可以接受一个类提供的数据和函数成员，也可以接受多个类提供的数据和函数成员，这两种形式分别称为单继承和多继承。

接受成员的新类称为派生类，而提供成员的类则称为基类。C ++为派生类提供了多种控制派生的方法：在基类的基础上增加新的成员；改变基类成员的访问权限；恢复基类成员的访问权限；重新定义和基类成员同名的成员。

基类私有成员是不能被包括派生类在内的其它任何函数成员访问的，除非将这些函数成员定义为基类的友元函数。基类的保护和公有成员可以 private、protected、public 等方式派生。派生类接受基类的数据和函数成员，在不同派生方式的控制作用下，在派生类提供的对外访问接口中，这些成员的访问权限就会有所不同。

当基类以 public 方式派生时，基类和派生类就构成父子关系，基类就可以被称为父类、派生类就可以被称为子类。父类指针可以直接指向子类对象，不需要进行强制类型转换。同理、父类引用也可以直接引用子类对象。

另一方面，在派生类的函数成员内部，无论基类是否使用 public 派生，基类和父类都被看作具有父子关系。在派生类的函数成员内，基类指针可以直接指向派生类对象，基类引用也可以直接引用派生类对象。

派生类除了接受基类的数据成员和函数成员，还可以定义自己的数据成员和函数成员，自定义的成员可以和基类成员同名。如果派生类的函数成员要调用基类的同名函数成员，则必须使用<基类名称>::<函数名称>的形式，否则，就可能出现派生类函数成员自递归调用的情形。因为根据面向对象的作用域，作用域小的标识符被访问的优先级更高，而基类成员的的作用域要比派生类成员的作用域大。

构造函数确定对象的初始状态，因此，了解构造函数的执行顺序非常重要。目前，由于所涉及的派生类都是单继承派生类，这样的派生类的构造函数的执行顺序比较容易确定。鉴于析构函数的执行顺序正好与构造函数的执行顺序相反，因此，本节仅介绍派生类的构造函数的执行顺序。

派生类构造函数的执行顺序是这样的：先调用虚基类的构造函数，接着调用基类的构造函数，然后按照数据成员的声明顺序，依次调用数据成员的构造函数或初始化数据成员，最后执行派生类构造函数的函数体。

如果虚基类或基类已定义了带参数的构造函数，或者派生类自己定义了引用成员、只

读成员或者需要用带参数的构造函数初始化的对象成员，则派生类必须定义自己的构造函数。虚基类或基类的构造函数总是会被调用的，不管它们是否出现在派生类初始化列表中。派生类只在构造虚基类或基类时调用它们的构造函数，此后，派生类不再调用或访问虚基类或基类的构造函数。

无论是全局变量还是局部变量，只要它们存储的是对象，在退出其作用域时，都会按照定义次序的相反顺序被自动析构。在执行变量的析构函数时，其内部执行顺序也正好和构造函数的执行顺序相反。

派生对象的存储单元包含基类的所有除静态数据成员外的数据成员。

三．习题答案

6.1 定义一个类 LOCATION，用数据成员 x、y 表示该类对象在二维坐标系中的坐标位置，用函数成员 move 移动对象坐标位置；然后，以类 LOCATION 为基类，派生出点类 POINT 和圆类 CIRCLE，定义点类和圆类相应的 move 和 draw 函数。

解：在 C++中，不能随意使用父类和子类这两个概念。当派生方式为 public 时，基类和派生类才能分别称为父类和子类。当两个类或多级派生的类构成父子关系时，父类指针可以直接指向子类对象，父类引用也可以直接引用子类对象，无须在指针或引用变量赋值时进行强制类型转换。如果没有构成父子关系，则必须进行进行强制类型转换。注意，在派生类的成员函数的函数体中，无论派生方式是否为 public，都认为基类和派生类构成父子关系。

```
#include <stdio.h>
class LOCATION{
    int    x, y;
public:
    int getx( ), gety( );
    void move(int x, int y);
    LOCATION(int x, int y);
};
void LOCATION::move(int x, int y)
{
    LOCATION::x=x;
    LOCATION::y=y;
}
int LOCATION::getx( ) { return x; }
int LOCATION::gety( ) { return y; }
LOCATION::LOCATION(int x, int y)
{
    LOCATION::x=x;
    LOCATION::y=y;
}
```

```
class POINT: public LOCATION{    //派生方式为 public，构成父子关系
    int visible;
public:
    int    isvisible( ) { return visible; };
    void draw( );
    void move(int x, int y);
    POINT(int x, int y) :LOCATION(x, y) {visible=0; };
};
void POINT::draw( )
{
    visible=1;
    printf("draw a point\n");
}
void POINT::move(int x, int y)
{
    visible=1;
    LOCATION::move(x, y);        //去掉 LOCATION::将造成自递归
    if(visible) draw( );
}
class CIRCLE: public POINT{      //派生方式为 public，构成父子关系
    int r;
public:
    int getr( ) { return    r; }
    void draw( ) { printf("draw a circle\n"); };
    CIRCLE(int x, int y, int r):POINT(x, y) { CIRCLE::r=r; };
};
void    main( ){
    LOCATION    *h;
    POINT           p(2, 3);
    CIRCLE          c(3, 4, 5);
    h=&p;                        //不需要强制类型转换：h=( LOCATION *)&p
    h=&c;                        //同上，public 派生使 POINT 和 CIRCLE 构成父子类
}
```

6.2 由点类派生出线段类，点类定义了函数成员 move 和 draw，请定义线段类的函数成员。

解：由于线段有两个端点，而单继承只能提供一个端点，故必须在类 LINE 中定义另一个端点，由于该端点是 POINT 类的对象，故这样的数据成员被称为对象成员。注意，对象成员 POIN 的初始化不能在 LINE 构造函数的函数体内进行。

```
class LINE: public POINT{        //构成父子关系
    POINT    end;                //定义对象成员
```

```
public:
    int getsx( ) { return   getx( ); }
    int getsy( ) { return   gety( ); }
    int getex( ) { return   end.getx( ); }
    int getey( ) { return   end.gety( ); }
    void draw( ) { printf("draw a line\n"); };
    void move(int x, int y){ POINT::move(x, y); end.move(x, y); };
    LINE(int sx, int sy, int ex, int ey):POINT(sx, sy), end(ex, ey) { };
};
```

6.3 定义描述个人信息的类 PERSON，记录个人姓名 name 和身份证号 IDnumber 等信息，定义函数成员 print，打印个人基本情况；然后由类 PERSON 派生出类 TEACHER，记录教师个人职称、工薪等信息；定义函数成员 print，打印教师的基本情况。

解：普通函数成员的调用是根据变量的的类型声明完成的。以下程序定义的指针 q 声明为指向 PERSON 类型，故无论 q 是否实际上指向 PERSON 的对象 p，还是指向 TEACHER 的对象 t，q->print( )调用的都是 PERSON 类型的普通函数成员 print( )。为了使对象的行为 print 与对象的类型 TEACHER 一致，必须将 print 函数定义为多态函数即虚函数，参见习题 7.1。

```
#include <stdio.h>
#include <string.h>
class PERSON{
    char name[10];
    char IDnumber[18];
public:
    void print( ){printf("Name=%s, IDnumber=%s\n", name, IDnumber);};
    PERSON(char *n, char *i);
};
PERSON::PERSON(char *n, char *i)
{
    strcpy(name, n);
    strcpy(IDnumber, i);
}
class TEACHER: public PERSON{   //派生方式 public，构成父子关系
    char title[20];
    double wage;
public:
    void print( );
    TEACHER(char *n, char *i, char *t, double w);
};
TEACHER::TEACHER(char *n, char *i, char *t, double w):PERSON(n, i) {
```

```
        strcpy(title, t);
        wage=w;
    };
    void TEACHER::print( ){
        PERSON::print( );
        printf("Title=%s, Wage=%d\n", title, wage);
    }
    void main( ){
        PERSON     *q, p("Liu", "420106641107538");
        TEACHER    t("Wei", "420106641107168", "Professor", 3000);
        q=&p; q->print( );    //输出了姓名和编号，q 定义为指向 PERSON 对象
        q=&t; q->print( );    //输出了姓名和编号，但 q 实际指向的是 TEACHER 对象
        t.print( );           //输出了姓名、编号、职称和工资，t 定义为 TEACHER 对象
    }
```

6.4 利用第四章练习中的 STACK，定义如下 REVERSE 类的函数成员，其构造函数将字符串压栈，其析构函数弹出字符并将其打印出来，打印的字符串正好是原字符串的逆序。然后，用 main 函数检验定义是否正确。

```
#include <stack.h>
class   REVERSE: STACK{
public:
    REVERSE(char *str);          //将字符串压栈
    ~REVERSE( );                 //按逆序打印字符串
};
void main(void) {
    REVERSE a("abcdefghij");
    REVERSE b("abcdefghijk");
}
```

解：注意，析构是构造的逆序，先定义的后析构，后定义的先析构。先执行输出的是 b 的析构函数，后执行输出的是 a 的析构函数。

```
#include <string.h>
#include <stack.h>                   //在此引用 STACK 的类型说明
class   REVERSE: STACK{
public:
    REVERSE(char *str);          //将字符串压栈
    ~REVERSE( );                 //按逆序打印字符串
};
REVERSE ::REVERSE(char *str):STACK(strlen(str)){
    for(int s=0, t=strlen(str); s<t; s++)    push(str[s]);
}
```

```
REVERSE ::~REVERSE( ){
        char    c;
        while(pop(c)) printf(“%c”, c);
}
void main(void) {
        REVERSE a("abcdefghij");
        REVERSE b("abcdefghijk");
}
```

6.5 对于如下类型声明，分别指出类 A、B、C 可访问的成员及其访问权限。

```
class A{
        int a1, a2;
protected:
        int a3, a4;
public:
        int a5, a6;
        A(int);
        ~A(void);
};
class B: A{
        int b1, b2;
protected:
        A::a3;
        int b3, b4;
public:
        A::a6;
        int b5, b6;
        B(int);
        ~B(void);
};
struct C: B{
        A::a5;
        int c1, c2;
protected:
        A::a4;
        int c3, c4;
public:
        int c5, c6;
        C(int);
        ~C(void);
```

};

解：注意构造函数是不能被直接调用的，也不能取构造函数的入口地址，构造函数是不可被访问的。私有成员除了能被类自己的函数成员访问外，不能被其他任何非友员函数或非友员函数成员访问。除非派生类函数定义为基类的友员，否则便不能访问基类的私有成员。

Class A 的可访问成员及其访问权限如下：

```
private:
    int a1, a2;
protected:
    int a3, a4;
public:
    int a5, a6;
    ~A( );
```

Class B 的可访问成员及其访问权限如下：

```
private:
    int  b1, b2, a4,a5;
    ~A( )
protected:
    int  a3, b3, b4;
public:
    int  a6, b5, b6;
    ~B(void);
```

Class C 的可访问成员及其访问权限如下：

```
protected:
    int  a3, a4, b3, b4, c3, c4;
public:
    int  a5, b5, b6, c1, c2, c5, c6;
    ~B(void);
    ~C(void);
```

6.6 如下程序定义了类 WINDOW 和类 MENU，由这两个类派生出下拉菜单类 PULLMENU 和弹出菜单类 POPMENU，并进一步派生出带下拉菜单的窗口类 PULLWIND 和带下拉及弹出菜单的窗口类 PULLPOPWIND。请补充和完善 PULLMENU、POPMENU、PULLWIND 和 PULLPOPWIND 的类型定义。

```
class WINDOW{
    char *title;                    //窗口的标题
    char *cover;                        //用于保存和恢复窗口覆盖的区域
    int  x, y;                          //窗口的左上角坐标
    int  w, h;                          //窗口的宽度和高度
    static WINDOW *head;                //窗口链，链首为最外层窗口
```

```
public:
    int    moveto(int x, int y);          //窗口移动到新的坐标
    void top( )const;                     //当前窗口成为顶层窗口
    void show( )const;                    //显示窗口，注意窗口的相互覆盖
    void hide( )const;                    //消隐窗口，注意窗口的相互覆盖
    WINDOW(char *n, int x, int y, int w, int h);
    ~WINDOW( );
};
class MENU{
    char *title;                          //菜单标题
    int    active;                        //菜单项是否可选的标志
    int (*entry)( );                      //该菜单项要执行的函数入口地址
public:
    int show( )const;                     //显示菜单项
    int isactive( )const;                 //判定菜单项是否可选
    int setactive(int a);                 //设置可选标志并返回设置前的标志
    int(*setentry(int(*)( )))( );         //设置新入口函数并返回旧入口函数的地址
    int execute( )const;                  //调用该菜单项的函数
    MENU(char *t, int a, int (*e)( ));
    ~MENU( );
};
class PULLMENU{
    MENU (&menus)[ ];                     //下拉菜单的各菜单项
    int    count;                         //下拉菜单的菜单项数
    //...                                 //其他数据成员和函数成员
};
class POPMENU{
    MENU (&menus)[ ];                     //下拉菜单的各菜单项
    int    count;                         //下拉菜单的菜单项数
    //...                                 //其他数据成员和函数成员
};
class PULLWIND: WINDOW{
    PULLMENU &pullm;
    //...                                 //其他数据成员和函数成员
};
class PULLPOPWIND: PULLWIND{
    POPMENU popm;
    //...                                 //其他数据成员和函数成员
};
```

解：本习题考察如何为类添加所需要的成员函数。以下仅列出相关的类型说明。

```
class PULLMENU{
    MENU (&menus)[ ];                    //下拉菜单的各菜单项
    int    count;                        //下拉菜单的菜单项数
public:
    int show( )const;                    //显示菜单
    MENU getmenu(int x);
    PULLMENU (MENU menus[ ]);
    ~ PULLMENU ( );
};
class POPMENU{
    MENU (&menus)[ ];                    //下拉菜单的各菜单项
    int    count;                        //下拉菜单的菜单项数
public:
    int show( )const;                    //显示菜单
    MENU getmenu(int x);
    POPMENU(MENU menus[ ]);
    ~POPMENU( );
};
class PULLWIND: WINDOW{
    PULLMENU &pullm;
public:
    int show( )const;                    //显示菜单
    PULLMENU getmenu( ){return pullm;};
    PULLWIND(char *n, int x, int y, int w, int h, PULLMENU pullm);
    ~PULLWIND( );
};
class PULLPOPWIND: PULLWIND{
    POPMENU popm;
public:
    int show( )const;                    //显示菜单
    POPMENU getmenu( ){return popm;};
    PULLPOPWIND(char *n, int x, int y, int w, int h, PULLMENU m, POPMENU p);
    ~PULLPOPWIND( );
};
```

第 7 章　虚函数

一．基本内容

虚函数、动态绑定、虚函数入口地址表首址，虚析构函数，纯虚函数，抽象类、类的存储空间。

二．学习要点

虚函数是一种能根据发出调用的对象的类型动态绑定相应类成员函数的多态函数。虚函数用保留字 virtual 声明，在程序运行的过程中，动态绑定是通过存储于对象之中的一个指针完成的，该指针指向该对象所属类的虚函数入口地址表的表首。

通常只需在基类中定义虚函数，派生类中原型相同的函数将自动成为虚函数。不管进行了多少级派生，自动成为虚函数这种特性将一直传递下去。对于派生类的非静态函数成员，只要其原型和基类乃至基类祖先的虚函数原型相同，这些函数成员就会自动成为虚函数。

虚函数具有隐含的 this 参数，而静态函数成员没有隐含的 this 参数，因此，虚函数不能同时定义为静态函数成员。由于被构造的对象类型总是确定的，编译时就能确定要调用的是哪个构造函数，因此，构造函数不需要表现出运行时多态特性，故 C++不允许构造函数定义为虚函数。但 C++允许定义虚析构函数，如此可在某些情况下可防止资源泄露。

父类指针可以指向父类对象，也可以指向子类对象，因此，父类指针指向的对象只能在运行时确定类型。析构函数可以通过父类指针调用，因此，析构函数必须能在运行时动态绑定，故 C++允许析构函数定义为虚函数。

不能同时用 virtual 和 friend 定义函数，因为 virtual 表示该函数是当前类的函数成员，而 friend 表示该函数不是当前类的函数成员。也不能同时用 virtual 和 static 定义函数，因为 virtual 表示该成员函数有 this 参数，而 static 表示该成员函数没有 this 参数。

虚函数可以声明为 inline 函数，也可以重载、缺省和省略参数，或者定义为其它类的成员友元。虚函数必须编译成可调用函数，声明为 inline 函数并不能保证 inline 成功。在重载虚函数时，虚函数的参数个数或类型必须有所不同，仅返回类型不同是不允许的。同普通函数成员一样，类体里定义的虚函数将自动成为内联函数。基类的虚函数定义为其它类的成员友元，并不意味着派生类原型相同的虚函数也自动成为该类的友元。

由于 union 既不能定义基类又不能定义派生类，不存在父类指针和父类引用的问题，在编译时就能确定对象的类型，故 union 的成员函数不需要动态绑定。因此，C++规定 union 类的成员函数不能定义为虚函数。

纯虚函数是不必定义函数体的特殊虚函数。在定义虚函数时，用函数体=0 表示定义的虚函数为纯虚函数。纯虚函数是特殊的虚函数，故同样有隐含的 this 参数，不能同时用 static 定义纯虚函数。构造函数不能定义为虚函数，故不能定义为纯虚函数；析构函数可以定义为虚函数，故也可定义为纯虚函数。

继承了基类的纯虚函数但未其定义函数体或者在当前类定义了新的纯虚函数，这样的类称为抽象类，抽象类是不能有实例对象的类，常常用作派生类的基类。在多级派生的过程中，如果到某个派生类为止，所有纯虚函数都全部定义了函数体，则该派生类就会成为具体类。

抽象类是不能有实例对象意味着：不能定义该类型的常量、变量、函数参数和返回类型，也不能通过 new 产生该类的实例对象。但可以定义抽象类指针和抽象类引用，因为这类指针和引用可以指向或者引用子类，而子类可能是允许定义对象实例的具体类。

无论是继承了基类的还是自定义了虚函数或者纯虚函数，当前类的对象都会包含一个指向该类虚函数入口地址表的指针。因此，在计算类的大小时，必须考虑该指针所占用的存储单元。

三．习题答案

7.1 从如下基类 base 派生多个类，每个类都定义 void isa( )函数，调用 void isa( )打印每个类的类名。

```
#include <iostream.h>
class    base {
public:
      base( ) { cout<<"Constructing base\n"; }
      virtual void isa( ) {cout<<"base\n";}
      ~base( ) { cout<<"Constructing base\n"; }
};
```

解：由于 isa 被定义为虚函数，在调用 p->isa( )时，将根据 p 实际指向的对象类型映射到相应类型的 isa( )函数，从而使函数的行为同对象类型一致。否则对象的行为同对象的类型不一致，参见习题 6.3。

```
//在此引用上述类型说明
class    derive: public base {//构成父子关系
public:
      derive( ) { cout<<"Constructing derive\n"; }
      virtual void isa( ) {cout<<"derive\n";}
      ~derive( ) { cout<<"Constructing derive\n"; }
};
class    clique: public base {//构成父子关系
public:
      clique ( ) { cout<<"Constructing clique \n"; }
      virtual void isa( ) {cout<<"clique\n";}
      ~ clique ( ) { cout<<"Constructing clique \n"; }
};
void main( ){
```

```
    base       *p, b;
    clique      c;
    derive      d;
    b.isa( );   c.isa( );     d.isa( );
    p=&b;     p->isa( );   //调用 base 的 isa( )
    p=&c;     p->isa( );   //调用 clique 的 isa( )
    p=&d;     p->isa( );   //调用 derive 的 isa( )
}
```

7.2 在异质链表中，每个节点的类型不要求相同，节点指针通常使用父类指针。以大学学生及教职人员为例，学生信息包括姓名、年龄、社会保险号、年级和平均成绩等，职员信息包括姓名、年龄、社会保险号和工资等，教授信息包括姓名、年龄、社会保险号、工资和研究方向等。为大学学生及教职人员建立一个异质链表，插入、删除和打印大学人员信息。

解：在以下程序中，异质链表插入、删除分别由构造和析构函数完成。异质链表存放于静态数据成员 head，C++的静态数据成员是用来存放类的总体信息的，如类的所有活动对象的个数，所有对象形成的链表的表头节点等，这正是静态数据成员的用法。

```
#include <string.h>
class   PERSON{
    char    *name, *SIDN;
    int       age;
    PERSON *next;
public:
    static PERSON *head;
    PERSON(char *name, char *SIDN, int    age);
    virtual void print( );
    virtual ~PERSON( );
};
PERSON *PERSON::head=0;
PERSON::PERSON(char *n, char *S, int    a) {
    if (name=new char[strlen(n)+1]) strcpy(name, n);
    if (SIDN=new char[strlen(S)+1]) strcpy(SIDN, S);
    next=head;
    head=this;
}
PERSON::~PERSON( ){
    PERSON *m, *n=head;
    if (head==0) { return; }
    if (name){delete name; name=0;}
    if (SIDN){delete SIDN; SIDN=0;}
```

```
        if(head==this) { head=next; return ; }
        while (n!=this && n!=0){ m=n; n=n->next; }
        m->next=this->next;
}
void PERSON::print( ) {
        printf("name=%s, SIDN=%s, age=%d\n", name, SIDN, age);
}
class    STUDENT: public PERSON{
        int        grade;
        double average;
public:
        STUDENT(char *name, char *SIDN, int age, int grade, double average);
        void print( );
        ~STUDENT( ){ };
};
STUDENT::STUDENT(char *n, char *S, int a, int g, double v):PERSON(n, S, a){
        grade=g;
        average=v;
}
void STUDENT::print( ) {
        PERSON::print( );
        printf("grade=%d, average=%d\n", grade,    average);
}
class    CLERK: public PERSON{
        double wage;
public:
        CLERK(char *name, char *SIDN, int age, double wage);
        void print( );
        ~CLERK( ){ };
};
CLERK::CLERK(char *n, char *S, int a, double w):PERSON(n, S, a){
        wage=w;
}
void CLERK::print( ) {
        PERSON::print( );
        printf("wage=%lf\n",wage);
}
class    PROFESSOR: public CLERK{
        char      *field;
```

```
public:
    PROFESSOR(char *name, char *SIDN, int age, double wage, char *field);
    void print( );
    ~PROFESSOR( ){ };
};
PROFESSOR::PROFESSOR(char *n, char *S, int a, double w, char *f):CLERK(n, S, a, w){
    if(field=new char[strlen(f)+1]) strcpy(field, f);
}
void PROFESSOR::print( ) {
    CLERK::print( );
    printf("field=%s\n", field);
}
void main( ){
    STUDENT s1("s1", "1", 1, 1, 90);
    STUDENT s2("s1", "2", 1, 1, 90);
    CLERK c1("c1", "1", 1, 1000);
    CLERK c2("c2", "2", 1, 1000);
    PROFESSOR p1("p1", "1", 1, 2000, "computer");
    PROFESSOR p2("p2", "2", 1, 2200, "computer");
    s1.print( );
    c1.print( );
    p1.print( );
    c1.~CLERK( );
    p1.~PROFESSOR( );
}
```

7.3 指出如下程序的错误之处及其原因：

```
class   A{
    int    a;
    virtual int f( );
    virtual int g( )=0;
public:
    virtual A( );
} a;
class B: A{
    long f( );
    int    g(int);
} b;
A *p=new A;
B *q=new B;
```

```
int    f(A, B);
A g(B &);
int h(B *);
```

解：程序的错误之处急错误原因如下。

(1) virtual A( )中去掉 virtual ，构造函数不能为虚函数。

(2) 不能定义变量 a，因为类 A 是抽象类

(3) long f( )中的 long 应改为 int，函数的原型和基类函数的名称、参数对应类型相同时，返回类型必须相同才能自动成为虚函数。

(4) 不能定义变量 b，因为类 B 是抽象类

(5) 不能产生对象 new A，因为类 A 是抽象类

(6) 不能产生对象 new B，因为类 B 是抽象类

(7) 不能为函数 int f(A, B)产生实参对象，因为类 A、B 是抽象类

(8) 函数 A g(B &)不能返回 A 类对象，因为类 A 是抽象类

7.4 在二维坐标系上定义 GRAPH 抽象类，该类具有基点坐标和图形显示等纯虚函数。从该类派生出三角形、圆等特定图形类，并为特定图形类定义相应纯虚函数。

解：在以下程序片段中，类 GRAPH 的纯虚函数的原型为 double area( )const=0。其中，const 限定所有自 GRAPH 派生的类的对象：在计算面积时不能改变当前对象的值。由于定义了纯虚函数，类 GRAPH 是个抽象类。

```
#include <math.h>
class GRAPH{
    double x, y;
public:
    double getx( )const{ return x; }
    double gety( )const{ return y; }
    virtual double area( )const=0;
    GRAPH(double x1, double y1) { x=x1; y=y1; }
};
class TRIANGLE: public GRAPH{
    double x2, y2, x3, y3;
public:
    double getx1( )const{ return getx( ); }
    double gety1( )const{ return gety( ); }
    double getx2( )const{ return x2; }
    double gety2( )const{ return y2; }
    double getx3( )const{ return x3; }
    double gety3( )const{ return y3; }
    double area( )const;
    TRIANGLE(double x1, double y1, double x2, double y2, double x3, double y3);
};
```

```
TRIANGLE::TRIANGLE(double x1, double y1, double x2, double y2, double x3, double y3)
    :GRAPH(x1, y1), x2(x2), y2(y2), x3(x3), y3(y3) { };
double TRIANGLE::area( )const
{
    double  a;
    a=(gety( )+y3)*(x3-getx( ));
    a=a+(gety( )+y2)*(getx( )-x2);
    a=a+(y2+y3)*(x2-x3);
    return  abs(a)/2;
}
class CIRCLE: public GRAPH{
    double r;
public:
    double getr( )const{ return r; };
    double area( )const;
    CIRCLE(double x,double y,double r):GRAPH(x, y) { CIRCLE::r=r; };
};
double CIRCLE::area( )const{ return 3.1415926536*r*r; }
```

7.5 为线性表定义一个抽象类，记录线性表的容量和当前元素个数，提供插入、删除、查找等纯虚函数。请从上述抽象类派生出线性表类，并用数组存放线性表元素。

解：注意 ABSL 是抽象类，但 LIST 不是抽象类，因为 LIST 定义类 ABSL 的所有纯虚函数。只要一个类没有基类遗留的未定义的纯虚函数，并且没有自己声明或定义新的纯虚函数，则该类就是一个具体类。

```
class ABSL{
    int numb;
    const int size;
public:
    int &getn( ){ return numb; };
    int gets( ){ return size; };
    virtual int insert(int v)=0;
    virtual int remove(int&v)=0;
    ABSL(int s):size(s){ numb=0; }
};
class LIST: public ABSL{
    int *a;
public:
    int insert(int v);
    int remove(int&v);
    LIST(int t):ABSL((a=new int[t])?t:0){ };
```

```
};
int LIST::insert(int v){
    int k, m, &c=getn( );
    if(c==gets( )) return 0;
    for(k=0; k<c && a[k]<=v; k++);
    for(m=c; m>k; m--) a[m]=a[m-1];
    a[m]=v;
    c++;
    return 1;
}
int LIST::remove(int&v){
    int k, c=getn( );
    if(c==0) return 0;
    for(k=0; k<c && a[k]!=v; k++);
    if(k==c) return 0;
    for(c--; k<c; k++) a[k]=a[k+1];
    return 1;
}
```

7.6 公司员工分临时和正式两种类型，只有正式工才能担任经理。每个员工都由一个经理主管，并由主管经理发放工资，只有老板才没有主管经理。试定义公司员工、经理和老板三个类。

解：在定义该类时，要注意员工和经理是互相依存的：每个员工都由一个经理主管，一个经理是一个正式员工。出现这种循环依赖时，必须使用前向引用声明，先声明其中一个类型再定义其他类型才能完成类型定义。

```
#include <iostream.h>
class MANAGER;   //前向引用声明
class CLERK{
    char     name[10];
    char     flag;    //职员正式和临时标志
    MANAGER    *monitor;
    double balance;
public:
    virtual double drawwage( );
    virtual double getbalance( );
    virtual void setmonitor(MANAGER *m);
    CLERK(char *n, MANAGER *m, double b, char f='t');
};
class MANAGER: public CLERK{
    char    title[10];
```

```
public:
    virtual double paywage( );
    MANAGER(char *n, char *t, MANAGER *m, double b);
};
double MANAGER::paywage( ){ double d; cin>>d; return d; }
MANAGER::MANAGER(char *n, char *t, MANAGER *m, double b):CLERK(n, m, b, 'f'){
    strcpy(title, t);
}
double CLERK::drawwage( ){ double d; balance+=monitor->paywage( ); return d; }
double CLERK::getbalance( ) {return balance; }
void   CLERK::setmonitor(MANAGER *m) { monitor=m; }
CLERK::CLERK(char *n, MANAGER *m, double b, char f){
    strcpy(name, n);
    monitor=m;
    balance=b;
    flag=f;
}
void main( ){
    MANAGER boss("Wang", "Tycon", 0, 100000);
    MANAGER man1("Yang", "Monitor", &boss, 20000);
    MANAGER man2("Zang", "Manager", &man1, 30000);
    CLERK    form("Cang", &man2, 1000, 'f');
    CLERK    temp("Tang", &man2, 1000, 't');
    CLERK    temp("Cang", &man1, 1000);
}
```

7.7 广义表的元素要么是单个字符，要么又是一个广义表。试定义广义表存放单个字符的类，该类提供一个打印字符的函数 print。由上述类派生出广义表类，同时定义插入函数 insert 和广义表打印函数 print。

解：注意广义表是可以自递归的或间接递归的，出现这种情况时将导致广义表打印函数陷入无穷递归。

```
#include <stdio.h>
class ELEMS{
    char     c;
    ELEMS    *n;
public:
    virtual void print( );
    ELEMS*getn( ){ return n; }
    ELEMS*setn(ELEMS *a){ n=a; return this; }
    ELEMS(char x, ELEMS *y=0){ c=x; n=y; }
```

```
};
void ELEMS::print( ){
    printf("%c", c);
}
class LISTS: public ELEMS{
    ELEMS    *e;
public:
    void print( );   //自动成为虚函数
    LISTS&join(ELEMS *e);
    LISTS(char x):ELEMS(x){ e=0; }
};
void LISTS::print( ){
    ELEMS::print( );
    printf("(");
    ELEMS *h=e;
    if(h) h->print( );
    while(h=h->getn( )){ printf(","); h->print( ); }
    printf(")");
}
LISTS&    LISTS::join(ELEMS *a){
    e=a->setn(e);
    return *this;
}
void main( ){
    ELEMS a('a'), b('b'), c('c'), d('d'), e('e');
    LISTS A('A'), B('B');
    A.join(&a); A.join(&b); A.join(&c);
    A.print( );
    B.join(&a); B.join(&A); B.join(&e);
    B.print( );
}
```

第 8 章　多继承类

一．基本内容

多继承、虚基类、访问二义性、构造顺序、存储空间。

二．学习要点

多继承是指在派生时有两个以上基类的继承。单继承是多继承的一个特例。在继承多个基类或虚基类成员的基础上，多继承派生类可以定义新的数据成员和函数成员，或者重新定义基类已有的数据成员或函数成员。

某些面向对象的语言只能定义单继承类。当需要描述多继承类型的对象时，常常通过对象成员或委托代理实现多继承。委托代理在多数情况下能够满足需要，但当多个对象实际上描述同一个物理对象时，就可能对该物理对象重复进行初始化。为此，C++专门引入虚基类，以便将多个基类的逻辑对象映射同一个物理对象。

虚基类用 virtual 声明。在派生类形成的派生森林中，同一棵派生树中的同名虚基类对象将被映射为同一个物理对象，无论有多少同名虚基类对象，该物理对象仅仅被构造函数初始化一次，而且尽可能早地初始化以供相关派生类使用。

在派生类中，除了派生类自己定义的成员外，还有继承自基类或虚基类的成员，它们之间很容易出现成员同名。如果派生类自己定义的成员和基类或虚基类的成员同名，根据面向对象的作用域规则，优先访问的应该是派生类自己定义的成员，其次是基类成员，最后是虚基类成员。如果同是基类或虚基类的成员同名，则应使用<基类名>::<成员名>显示指定要访问的成员，这样便可以避免出现访问二义性问题。

在多继承派生的情况下，尤其是当类的继承关系比较复杂时，类的构造与析构顺序更加难于把握。但是，类的构造与析构顺序对于理解对象的行为至关重要。本节仅讨论多继承派生类构造函数的执行顺序，析构函数的执行顺序正好同构造函数的执行顺序相反。

在考虑派生类构造函数的执行顺序时，必须注意派生类可能有虚基类、基类、对象成员、const 成员以及引用成员。当虚基类、基类和对象成员的构造函数带参数时，派生类必须定义自己的构造函数，而不能利用 C++提供的缺省构造函数。

对于虚基类、基类和对象成员来说，如果它们没有定义自己的构造函数，则编译程序就会为它们提供缺省的无参构造函数。对于虚基类、基类和对象成员的构造函数，无论它们是自定义的还是由编译程序提供的，都会被派生类构造函数按某种顺序明确或隐式地调用。

派生类对象的构造顺序比较复杂：①按定义顺序自左至右、自下而上地构造所有虚基类；②按定义顺序构造派生类的所有直接基类；③按定义顺序构造派生类的所有数据成员，包括对象成员、const 成员和引用成员；④执行派生类自身的构造函数体；⑤如果虚基类、基类、对象成员、const 成员以及引用成员又是派生类对象，重复上述派生类对象的构造过程，但同名虚基类对象在同一棵派生树中仅构造一次。

在派生类没有虚基类的情况下，计算派生类的存储空间比较容易。如果派生类的第一个基类建立了虚函数地址表，则派生类就共用该表首址所占用的存储单元；如果派生类的第一个基类没有定义虚函数，派生类就在建立完所有基类的存储空间之后，根据派生类是否建立了虚函数入口地址表，确定是否为该表首址分配一个存储单元，然后为新定义的数据成员的建立存储空间。

在派生类有虚基类的情况下，虚基类的存储空间建于派生类的尾部，虚基类的存储空间按虚基类的构造顺序建立。派生类建立存储空间的次序如下：①派生类依次处理每个直接基类或虚基类，如果为直接基类，则为其建立存储空间，如果为直接虚基类则建立一个到虚基类的偏移。②如果派生类继承的第一个类为基类，且该基类定义了虚函数地址表，则派生类就共享该表首址占用的存储单元。对于除前述情形外的其他任何情形，派生类在处理完所有基类或虚基类后，根据派生类是否建立了虚函数地址表，确定是否为该表首址分配存储单元。③派生类依次处理自定义的数据成员，为每个数据成员建立相应的存储空间。④派生类根据虚基类偏移的建立顺序，依次为虚基类建立存储空间，同名虚基类仅在派生类存储空间内建立一次。⑤如果直接基类和虚基类又是派生类，则在派生类的存储空间内重复步骤①至⑤。如果数据成员又为派生类类型，则在数据成员的存储空间内重复步骤①至⑤。

由此可见，派生类在有静态数据成员、虚函数、虚基类的情况下，其对象的存储空间布局是非常复杂。关于空间布局问题不要求完全掌握，但对某些系统开发人员来说，掌握多继承派生类的空间布局是十分必要的。

三．习题答案

8.1 定义两个类：

```
class fruit {
public:
    virtual char *identify( ) { return    “fruit”; }
};
class tree{
public:
    virtual char *identify( ) { return    “tree”; }
};
```

请派生出一些既是 fruit 又是 tree 的类及其派生类。例如，apple 类：

```
class apple: public fruit, public tree{
    //…
}
```

为每个派生类定义一个函数成员，该函数成员显示自己的类名，并显示类由哪些基类派生。例如，apple 类的函数成员显示如下信息：

(apple: fruit, tree)

苹果梨 apple_pear 类的函数成员显示如下信息：

(apple_pear: (apple: fruit, tree)，(pear: fruit, tree))

解：多重派生时，通常要注意虚函数的使用，要注意在基类中定义虚函数，则派生类的原型相同的函数成员，即使不使用 virtual 声明也会自动成为虚函数。

```
class Apple: public fruit, public tree{
public:
    char *identify( ) {
        static char s[40];
        strcpy(s, "(apple: ");
        strcat(s, fruit::identify( ));
        strcat(s, ", ");
        strcat(s, tree::identify( ));
        strcat(s, ")");
        return  s;
    };
};
class Pear: public fruit, public tree{
public:
    char *identify( ) {
        static char s[40];
        strcpy(s, "(pear: ");
        strcat(s, fruit::identify( ));
        strcat(s, ", ");
        strcat(s, tree::identify( ));
        strcat(s, ")");
        return  s;
    };
};
class ApplePear: public Apple, public Pear{
public:
    char *identify( ) {
        static char s[60];
        strcpy(s, "(apple pear: ");
        strcat(s, Apple::identify( ));
        strcat(s, ", ");
        strcat(s, Pear::identify( ));
        strcat(s, ")");
        return  s;
    };
};
```

8.2 指出如下各类可访问的成员及成员的访问权限。

```
class A{
    int  a;
protected:
    int  b, c;
public:
    int  d, e;
    ~A( );
};
class B: protected A{
    int  a;
protected:
    int  b, f;
public:
    int  e, g;
    A::d;
};
class C: A{
    int  a;
protected:
    int  b, f;
public:
    int  e, g;
    A::d;
};
struct D: B, C{
    int  a;
protected:
    int  b, f;
public:
    int  e, g;
};
```

解：派生类要继承基类的数据成员和函数成员，但继承的成员并不是都是可以访问的，例如基类的构造函数成员就是不可访问的。各类可访问的成员及成员的访问权限如下：

类 A 的可访问成员及访问权限：

```
private:
    int  a;
protected:
    int  b, c;
public:
```

```
    int    d, e;
    ~A( );
```
类 B 的可访问成员及访问权限：
```
private:
    int    a;
protected:
    int    b, A::b, c, A::e, f;
    ~A( );
public:
    int    A::d , e, g;
```
类 C 的可访问成员及访问权限：
```
private:
    int    a, A::b, c;
    ~A( );
protected:
    int    b, f;
public:
    int    A::d , e, g;
```
类 D 的可访问成员及访问权限：
```
protected:
    int    B::b, C::b, b, c, B::f, C::f,f;
    ~A( );
public:
    int    a, B::d, C::d, B::e, C::e, e, B::g, C::g, g;
```

8.3 指出如下程序中 main 的每行语句的输出结果。

```
#include <iostream.h>
struct A{A( ) { cout<<'A';}};
struct B{B( ) { cout<<'B';}};
struct C: A{C( ) { cout<<'C';}};
struct D: A, B{D( ) { cout<<'D';}};
struct E: A, B, virtual C{
    D    d;
    E( ) { cout<<'E';}
};
struct F: A, virtual B, virtual C, D, E{
    C    c, d;
    E    e;
    F( ) { cout<<'F';}
};
```

```
void main(void)
{
    A  a;  cout<<'\n';
    B  b;  cout<<'\n';
    C  c;  cout<<'\n';
    D  d;  cout<<'\n';
    E  e;  cout<<'\n';
    F  f;  cout<<'\n';
}
```

解：在同一棵派生树中，同名虚基类只构造一次，且要注意虚基类最先构造，若有多个虚基类则按宽度优先搜索的到顺序构造。其次是按定义顺序构造所有的基类，然后是按定义顺序初始化或构造所有的数据成员，最后执行构造函数自己的函数体。虽然本题只要求给出构造的顺序，但要注意析构是构造的逆叙。各行的输出见如下注解：

```
void main(void)
{
    A  a;  cout<<'\n';        //输出 A
    B  b;  cout<<'\n';        //输出 B
    C  c;  cout<<'\n';        //输出 AC
    D  d;  cout<<'\n';        //输出 ABD
    E  e;  cout<<'\n';        //输出 ACABABDE
    F  f;  cout<<'\n';        //输出 BACAABDABABDEACACACABABDEF
}
```

8.4 定义一个圆和一个三角形类，它们分别有构造函数、析构函数和绘图函数。从圆和三角形类派生出圆内接三角形类，定义圆内接三角形类的数据和函数成员。

解：已知三角形的三个顶点坐标(x1, y1), (x2,y2), (x3,y3),可得起外接圆的圆心(x,y)满足：

$$\begin{cases}(x1-x)^2+(y1-y)^2=(x2-x)^2+(y2-y)^2\\(x1-x)^2+(y1-y)^2=(x3-x)^2+(y3-y)^2\end{cases}$$

得到：

$$X=\frac{(x1*x1*y2+x2*x2*y3+x3*x3*y1+y1*y1*y2+y2*y2*y3+y3*y3*y1-x1*y3*y3-x2*y1*y1-x3*y2*y2-y1*y1*y3-y2*y2*y1-y3*y3*y2)}{2(x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)}$$

$$Y=\frac{(x1*x2*x2+x2*x3*x3+x3*x1*x1+x1*y2*y2+x2*y3*y3+x3*y1*y1-x1*y3*y3-x2*y1*y1-x3*y2*y2-x1*x3*x3-x2*x1*x1-x3*x2*x2)}{2(x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)}$$

由此可得圆的半径公式：r=sqrt(pow(x1-x, 2)+pow(y1-y,2))。程序如下：

```
#include <iostream.h>
#include <math.h>
class CIRCLE{
    double x, y, r;
```

```
public:
    double getx( ) { return x; };
    double gety( ) { return y; };
    double getr( ) { return r; };
    CIRCLE(double x, double y, double r):x(x), y(y), r(r) { };
    void draw( ) { cout<<"draw circle at("<<x<<","<<y<<") with radius "<<r<<"\n";};
};
class TRIANGLE{
    double x1, y1, x2, y2, x3, y3;
public:
    double getx1( ) { return x1; };
    double gety1( ) { return y1; };
    double getx2( ) { return x2; };
    double gety2( ) { return y2; };
    double getx3( ) { return x3; };
    double gety3( ) { return y3; };
    TRIANGLE(double x1, double y1, double x2, double y2, double x3, double y3)
        :x1(x1), y1(y1), x2(x2), y2(y2), x3(x3), y3(y3) { };
    void draw( ) {
        cout<<"draw triangle at("<<x1<<","<<y1<<"),(";
        cout<<x2<<","<<y2<<"),("<<x3<<","<<y3<<")";
    }
};
class INSCRIBED:CIRCLE, TRIANGLE{
public:
    INSCRIBED (double x1, double y1, double x2, double y2, double x3, double y3);
    void draw( ) {
        cout<<"draw inscribed triangle at("<<getx1( )<<","<<gety1( ) <<"),(";
        cout<<getx2( )<<","<<gety2( )<<"),("<<getx3( )<<","<<gety3( )<<")";
    }
};
INSCRIBED::INSCRIBED (double x1,double y1,double x2,double y2,double x3,double y3)
:CIRCLE(
        (x1*x1*y2+x2*x2*y3+x3*x3*y1+y1*y1*y2+y2*y2*y3+y3*y3*y1-x1*y3*y3
            -x2*y1*y1-x3*y2*y2-y1*y1*y3-y2*y2*y1-y3*y3*y2)/
            (x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)/2,
        (x1*x2*x2+x2*x3*x3+x3*x1*x1+x1*y2*y2+x2*y3*y3+x3*y1*y1-x1*y3*y3
            -x2*y1*y1-x3*y2*y2-x1*x3*x3-x2*x1*x1-x3*x2*x2)/
            (x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)/2,
```

```
        sqrt(pow(x1-(x1*x1*y2+x2*x2*y3+x3*x3*y1+y1*y1*y2+y2*y2*y3+y3*y3*y1
                -x1*y3*y3-x2*y1*y1-x3*y2*y2-y1*y1*y3-y2*y2*y1-y3*y3*y2)/
                (x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)/2,2)
            +pow(y1-(x1*x2*x2+x1*y2*y2+x2*x3*x3+x2*y3*y3+x3*x1*x1+x3*y1*y1
                -x1*x3*x3-x1*y3*y3-x2*x1*x1-x2*y1*y1-x3*x2*x2-x3*y2*y2)/
                (x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)/2,2))
), TRIANGLE(x1, y1, x2, y2, x3, y3){ }
```

8.5 定义一个显示器 DISPLAY 类和一个键盘 KEYBOARD 类，并由此派生出反馈显示输入字符的控制台 CONSOLE 类。DISPLAY 类有字符输出 int putc(char)函数，KEYBOARD 类有不反馈显示输入字符的字符输入 int getch(void)函数。

解：在以下程序中，CONSOLE 类的 getch 函数调用了基类 KEYBOARD 的 getch 函数。调用时一定要用 KEYBOARD：：getch( )这样的形式，若去掉 getch 的作用域限定形式 KEYBOARD：：，则导致 CONSOLE 类的 getch 函数调用 CONSOLE 自己的 getch 函数，从而造成无穷递归。根据面向对象的作用域规则，在成员函数中优先访问本类的成员。

```
class DISPLAY{
public:
    int    putc(char c);
};
class KEYBOARD{
public:
    int getch(void);
};
class CONSOLE: DISPLAY, KEYBOARD{
public:
    int getch(void){    return putc(KEYBOARD::getch( ));    } ;
};
```

8.6 将带下拉和弹出窗口的类 PULLPOPWIND 设计成一个多继承类。

解：程序如下：

```
class PULLMENU{
    MENU (&menus)[ ];                  //下拉菜单的各菜单项
    int    count;                      //下拉菜单的菜单项数
public:
    int show( )const;                  //显示菜单
    MENU getmenu(int x);
    PULLMENU (MENU menus[ ]);
};
class POPMENU{
    MENU (&menus)[ ];                  //下拉菜单的各菜单项
    int    count;                      //下拉菜单的菜单项数
```

```
public:
    int show( )const;                    //显示菜单
    MENU getmenu(int x);
    POPMENU(MENU menus[ ]);
};
class PULLPOPWIND: WINDOW, PULLMENU , POPMENU {
public:
    int show( )const;                    //显示菜单
    PULLMENU getmenu( ){return pullm;};
    PULLPOPWIND(char *n, int x, int y, int w, int h, PULLMENU p, POPMENU m);
};
```

第 9 章　运算符重载

一．基本内容

左值、右值、左值运算符、右值运算符，单目、双目、三目、任意运算符、浅拷贝赋值、深拷贝赋值、类型转换运算符重载。

二．学习要点

左值表达式是能出现在等号左边的表达式、右值表达式是能出现在等号右边的表达式。C++的某些运算符运算的结果仍然为左值，这样的运算符称为左值运算符。左值运算符有：前置++和—运算，以及各种类型的赋值运算等。

只有一个操作数的运算符称为单目运算符，有两个操作数的运算符称为双目运算符，有三个操作数的运算符称为三目运算符，有任意个操作数的运算符称为任意目运算符。有些运算符既可作单目运算符，又可作双目运算副。可作三目运算符的运算符只有一个即**? :**，圆括号括起参数表可看作任意目运算符，圆括号括起类型表达式可看作类型转换运算符。

除运算符 sizeof、. 、.*、::和三目运算符? :外，所有运算符函数都可以重载。运算符＝、–>、( )、[ ]只能重载为普通函数成员，不能重载为静态函数成员或普通函数；运算符 new 和 delete 不能重载为普通函数成员，可以重载为静态函数成员或普通函数；其他运算符都不能重载为静态函数成员，但可以重载为普通函数成员和普通函数。

运算符重载是面向类的单个对象的，而不是面向简单类型的变量或常量的。如果将运算符函数重载为普通函数，就必须至少定义一个类型为类或者类的引用的参数，而不能将参数定义为对象指针或者对象数组类型，后者不能用来代表类的单个对象。

重载运算符时，要注意运算符是否为左值运算符，同时要注意第一个参数是否要求为左值。如果运算符是左值运算符，则重载后运算符函数最好返回引用类型，这样便可以连续的进行运算。例如，整型变量 x 就可以连续进行运算(x=3)=8；因为 x=3 的运算结果为左值，可以继续出现在=8 的左边。

重载不改变运算符的优先级和结合性。在一般情况下，重载也不改变运算符的操作数个数，即不改变运算符函数的参数个数。但是，纯单目的运算符++和—存在着前置和后置两种运算，需要通过改变运算符的操作数个数来区分这两种运算；双目运算符–>重载为指针或引用也会改变运算符的操作数个数。

为了区分前置运算和后置运算，必须使它们的函数原型有所不同，必须将前置运算重载为单目运算符函数，将后置运算重载为双目运算符，这样可以避免出现函数重定义错误。

如果后置运算++和—重载为类的函数成员，则该函数成员除了有隐含的 this 参数外，还必须声明一个 int 类型的参数；如果后置运算++和—重载为普通函数，则该普通函数必须声明类和 int 类型的两个参数。

重载时应遵守 C++的习惯或约定，前置运算应该是先运算后取值，后置运算应该是先取值后运算；此外，前置运算重载函数应返回一个引用类型，表示该运算的结果是左值，可

以继续出现在等号的左边，例如++x=8。

许多运算都不应该改变操作数的值。例如，x+y 就不应该改变 x 和 y 的值，故类 A 的加法运算符可重载如下三种推荐形式，普通成员函数的原型应该为 A opertor+(const A&)const，静态成员函数和普通函数的原型为 A operator+(const A&, const A&)。

当类包含指针类型的成员时，必须重载拷贝构造函数和等号运算符，以便进行深拷贝构造和赋值操作。如果要定义的类将作为其他类的基类，并且希望通过父类指针调用多态析构，则基类的析构函数就应该声明为虚函数，其他普通函数成员也尽可能地定义为虚函数，这样定义的含指针成员的类才是一个安全的类。具体来说，定义包含指针类型的成员的类 T 应注意以下几点：

① 应定义 T(const T &)等形式的深拷贝构造函数，以解决值参传递浅拷贝问题；

② 应定义 virtual T &operator=(const T &)等形式的深拷贝赋值运算函数；

③ 应定义 virtual ~T( )形式的虚析构函数，解决析构内存泄露问题；

④ 在定义引用 T &p=*new T( )后，要使用 delete　&p 释放对象占用的内存;

⑤ 在定义指针 T *p=new T( )后，要使用 delete　p 释放对象占用的内存。

单参数的构造函数相当于类型转换函数。例如，假定类 A 定义了 A(int)构造函数，则类 A 的对象 a 可以赋值为 a=8，上述赋值语句的等价形式为 a=A(8)，正数 8 被自动地转换为一个对象常量 A(8)。类型转换函数运算符可减少其它运算符重载的个数。例如，只要重载了普通成员函数 A opertor +(const A&)，就不需要重载普通成员函数 A opertor +(int)了，因为表达式 a+8 可以被自动转换为 a+A(8)，从而就可以调用 A opertor +(const A&)了。

类型转换运算符( )只能重载为普通函数成员。例如，在类 A 中，其重载函数的推荐形式为 operator <类型表达式>( )const，注意：(1) const 表示将当前对象转换为<类型表达式>所代表的类型时不改变当前对象的值；(2) 类型转换运算函数没有返回类型，<类型表达式>所代表的类型就是其返回类型。

三．习题答案

9.1 指出以下程序中的语法错误及其原因。

```
class A{
    static const int a;
    int   b;
protected:
    int   c, d;
    int   A(int);
public:
    int   e, f;
    const int g=7;
    A &operator=(A &);
} a={1, 2, 3,};
class B: protected A{
    int   b;
```

```
protected:
    A::a;
    B(int);
    int    operator ? : (int, int);
public:
    static B(int, int);
    int operator int( );
    operator A( );
    B(int, int, int);
} b(5);
void main( void) {
    int    x;
    int    &y=x;
    int    &*z=&y;
    int    A::*p=&A::b;
    int    *q=&A::e;
    int    A::**r;
    x=b.b;
    x=b.f;
    y=x+a;
    y=B(6);
    a=B(3, 4, 5);
    r=&q;
    a.*q=6;
    *(a.*r)=9;
}
```

解：错误及其原因如下：

(1) int A(int)中，构造函数没有返回类型，不能返回 int。

(2) const int g=7 中，不能对 g 初始化，只读变量只能在构造函数的函数头后面用冒号隔开初始化。

(3) a={1, 2, 3,}中，因类 A 有私有成员或构造函数，必须使用构造函数初始化。

(4) A::a 是私有的，不能出现在 protected 下面。

(5) operator ?:不能重载?:运算符。

(6) static B(int, int)中不能出现 static。

(7) int operator int( )类型转换重载函数不能有返回类型。

(8) b(5)无法用私有的构造函数 B(int)初始化，全局变量只能有公有构造函数初始化。

(9) int &*z 不能定义指向引用类型的指针，引用类型的变量理论上不分配内存，此外初始化表达式的类型也不匹配。

(10) 在 main 中不能访问私有的成员 A::b。

(11) &A::e 的类型为 int A::*，不能转换为 int *。

(12) 在 x=b.b 中，B::b 是 main 不可访问的成员。

(13) 在 x=b.f 中，A::f 是 main 不可访问的成员。

(14) x+a 没有相应的运算符重载函数，也没有将类 A 的对象转换为 int 的函数。

(15) B(6)要构造一个常量对象，但保护的构造函数 B(int)是不能被 main 访问的。

(16) B 类没有类型转换转换常量对象 B(3, 4, 5) 为 A 类对象，A 类也没有相应赋值运算符函数 A::operator =(B)重载，不能实现类型转换后赋值或不同类型直接赋值。

(17) r=&q 中，&q 的类型为 int **，不能转换为和 r 同样的 int A::**类型。

(18) 在 a.*q 中，q 不是普通成员指针类型。

(18) 在 a.*r 中，r 是一个指向普通成员指针的普通指针，而不是普通成员指针。

9.2 指出 main 的变量 i 在每条语句执行后的值。

```
int    x=7;
int    y=::x+5;
struct A{
      int x;
      static int y;
public:
      A &operator +=(A &);
      operator int( ) { return x+y; };
      A(int x=::x+1, int y=::y+11) {
            A::x=x;
            A::y=y;
      }
};
A& A::operator +=(A &a) {
      x+=a.x;
      y+=a.y;
      return *this;
}
int A::y=20;
void main(void) {
      A    a(2, 5), b(6), c;
      int    i, &j=i,*p=&A::y;
      i=b.y;
      j=b.x;
      i=*p;
      j=c;
      i=a+c;
      i=b+=c;
```

```
    i=((a+=c)=b)+9;
}
```

解：注意静态数据成员 static int y 只有一个副本，为当前类所有对象共享内存，任何对象对 y 的操作都会影响所有其他对象。另外，要注意引用变量以及返回引用类型的函数成员 A& A::operator +=(A &a)，逻辑上引用是不分配内存的，所有对引用的赋值操作都是对被引用变量的赋值操作。变量 i 在 main 中每条语句执行后的值如下：

```
(1) i=b.y;              //i=23
(2) j=b.x;              //i=6
(3) j= *p;              //i=23
(4) j=c;                //i=31
(5) j=a+c;              //i=56
(6) i=b+=c;             //i=60
(7) i=((a+=c)=b)+9;     //i=115
```

9.3 定义如下复数类型 COMP 中的函数成员。

```
class  COMP{
    double  r, v;
public:
    COMP(double  r=0, double v=0);
    COMP operator + (const  COMP &)const;
    COMP operator – (const  COMP &)const;
    COMP operator * (const  COMP &)const;
    COMP operator / (const  COMP &)const;
};
```

解：本题构造函数的尾部没带 const，表明该函数的隐含参数的类型为 COMP*const this，即 this 指向的当前对象是可以修改或赋值的；其他函数成员的尾部带有 const，表明函数的隐含参数的类型为 const COMP*const this，即 this 指向的当前对象是不能修改或赋值的。注意大部分函数的返回类型为 COMP，必须定义新的对象变量或使用对象常量作为返回值；若函数的返回类型为 COMP&，则通常用 return *this 返回，表示返回值是一个左值，即函数调用可以出现在等号左边，当前操作完成后还可进行其他操作。例如，如果 operator +返回 COMP&且 a、b 是 COMP 类的对象，则可以书写这样的表达式 a+b=a。函数成员定义如下：

```
COMP::COMP(double  r, double v):r(r), v(v){ }
COMP COMP::operator + (const COMP &s)const{ return COMP(r+s.r, v+s.v); }
COMP COMP::operator - (const COMP &s)const{ return COMP(r-s.r, v-s.v); }
COMP COMP::operator * (const COMP &s)const{return COMP(r*s.r-v*s.v, r*s.v+v*s.r); }
COMP COMP::operator / (const  COMP &s)const{
    double d=s.r*s.r+s.v*s.v;
    return COMP((r*s.r+v*s.v)/d, (s.r*v-s.v*r)/d);
}
```

9.4 定义如下字符串类 STRING 的函数成员。

```
#include <string.h>
class STRING{
    char    *str;
public:
    STRING(char *s=0);
    STRING(const    STRING &);
    virtual    int    strlen(void)const;
    virtual    int    strcmp(const STRING &)const;
    virtual    STRING operator + (const STRING &)const;
    virtual    STRING &operator = (const STRING &);
    virtual    STRING &operator += (const STRING &);
    virtual    operator const char *(void)const;
    virtual    ~STRING(void);
};
```

解：参见上一题答案，注意 operator =的返回类型为 STRING&，意味在主函数中，可以出现(a=b)= " abcd"的表达式，即将 b 赋值给 a 后再将 STRING("abcd ")赋值给 a。函数成员定义如下：

```
STRING::STRING(char *s){    if(str=new char[::strlen(s)+1]) strcpy(str, s); }
STRING::STRING(const STRING &s)
{    if(str=new char[::strlen(s.str)+1])    strcpy(str, s.str);    }
int STRING::strlen(void)const{    return ::strlen(str);    }
int STRING::strcmp(const STRING &s)const{    return ::strcmp(str, s.str);    }
STRING STRING::operator +(const STRING &s)const{
    char *t=new char[::strlen(str)+::strlen(s.str)+1];
    STRING r(strcat(strcpy(t, str),s.str));
    delete [ ]t;
    return r;
}
STRING &STRING::operator=(const STRING &s){
    delete [ ]str;
    strcpy(str=new char[s.strlen( )+1], s.str);
    return *this;
}
STRING &STRING::operator += (const STRING &s){    return *this=*this+s;    }
STRING::operator const char *(void)const{    return str;    }
STRING::~STRING(void){    if(str) {delete str; str=0; }    }
void main( ){
    STRING    a("abcde"), b("fgh"), c(a);
```

```
    c+=b;
    (a=b)="abcd";
}
```

9.5 定义一个位数不限的十进制整数类 DEC，完成加、减、乘、整除、求余等运算。该类定义如下，试定义其中的函数成员。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <exception.h>
class DEC{
    char *n;     //存放十进制数
public:
    DEC(long);
    DEC(char *d);
    DEC(DEC const &d);
    virtual ~DEC( );
    virtual DEC operator-( )const;
    virtual DEC operator+(const DEC &d) const;
    virtual DEC operator-(const DEC &d) const;
    virtual DEC operator*(const DEC &d) const;
    virtual DEC operator/(const DEC &d) const;
    virtual DEC operator%(const DEC &d)const;
    virtual int operator>(const DEC &d)const;
    virtual int operator<(const DEC &d)const;
    virtual int operator==(const DEC &d)const;
    virtual DEC &operator=(const DEC &d);
    virtual DEC operator++(int);
    virtual DEC operator--(int);
    virtual DEC &operator--( );
    virtual DEC &operator++( );
    virtual operator const char *( )const;
};
```

解：这是一个十分有用的习题，可以用来计算无限大的整数。注意两个 operator++分别表示前置和后置运算，并注意它们的返回类型不同。在 C++中前值运算的结果为左值，返回类型为 DEC&，当前结果可以再次出现在等号左边，例如可以出现类似++d="10"之类的表达式。另外，要注意 operator++是用 operator+定义的，要注意前置运算是先运算后取值。同理，可以定义 operator<=等运算也可以用已有的运算定义。函数成员定义如下：

```
DEC::DEC(long m){
    char s[100];
```

```
    ltoa(m, s, 10);
    DEC::n=new char[strlen(s)+1];
    strcpy(DEC::n, s);
}
DEC::DEC(char *n){
    DEC::n=new char[strlen(n)+1];
    if(n[0]=='+') strcpy(DEC::n, n+1);
    else{
        if(n[0]=='-'&&n[1]=='0') strcpy(DEC::n, n+1);
        else strcpy(DEC::n, n);
    }
}
DEC::DEC(const DEC &d){
    if(n=new char[strlen(d.n)+1]) strcpy(n, d.n);
}
DEC::~DEC( ){ if(n) { delete n; n=0; } }
DEC DEC::operator-( )const{
    char *t=new char [strlen(n)+2];
    if(n[0]=='-') strcpy(t, n+1);
    else strcat(strcpy(t,"-"),n);
    DEC r(t);
    delete t;
    return r;
}
DEC &DEC::operator=(const DEC &d){
    delete n;
    if(n=new char[strlen(d.n)+1]) strcpy(n, d.n);
    return *this;
}
int DEC::operator>(const DEC &d)const{
    int h=strlen(n),k=strlen(d.n);
    switch(n[0]){
    case '-':
        switch(d.n[0]){
        case '-':
            if(h==k){
                for(k=0; k<h; k++){
                    if(n[k]==d.n[k]) continue;
                    if(n[k]<d.n[k]) return 1;
```

```
                    return 0;
                }
                return 0;
            }
            if(h<k) return 1;
            return 0;
        default :
            return 0;
        };
    default :
        switch(d.n[0]){
        case '-':
            return 1;
        default :
            if(h==k){
                for(k=0; k<h; k++){
                    if(n[k]==d.n[k]) continue;
                    if(n[k]<d.n[k]) return 0;
                    return 1;
                }
                return 0;
            }
            if(h<k) return 0;
            return 1;
        }
    }
}
int DEC::operator< (const DEC &d)const{ return d>*this || d==*this; }
int DEC::operator==(const DEC &d)const{ return strcmp(n, d.n)==0;}
DEC DEC::operator+(const DEC &d) const{
    switch(n[0]){
    case '-':
        switch(d.n[0]){
        case '-':
            return -(-*this+(-d));
        default :
            return d-(-*this);
        };
    default :
```

```
            switch(d.n[0]){
            case '-':
                return *this-(-d);
            default :
                int m, c=0, p=strlen(n), q=strlen(d.n);
                 if(p>q) m=p; else m=q; m+=2;
                char *t=new char[m]; t[--m]=0;
                while (p>0 && q>0){
                    c=c+n[--p]+d.n[--q]-2*'0';
                    t[--m]=((c>=10)?c-10: c)+'0';
                    c=(c>=10)? 1: 0;
                }
                while (p>0){
                    c=c+n[--p]-'0';
                    t[--m]=((c>=10)?c-10: c)+'0';
                    c=(c>=10)?1: 0;
                }
                while (q>0){
                    c=c+d.n[--q]-'0';
                    t[--m]=((c>=10)?c-10: c)+'0';
                    c=(c>=10)?1: 0;
                }
                if(c>0) {t[--m]='1'; }
                DEC x(t+m);
                delete t;
                return x;
            };
        };
}
DEC DEC::operator-(const DEC &d) const{
        switch(n[0]){
        case '-':
            switch(d.n[0]){
            case '-':
                return (-d)-(-*this);
            default :
                return -(-*this+d);
            };
        default :
```

```
            switch(d.n[0]){
            case '-':
                return *this+(-d);
            default :
                if(d>*this) return -(d-*this);
                int m, k, c=0, p=strlen(n), q=strlen(d.n);
                 if(p>q) m=p; else m=q; k=m+=2;
                char *t=new char[m]; t[--m]=0;
                while (p>0 && q>0){
                    c=n[--p]-d.n[--q]-c;
                    t[--m]=((c<0)?c+10: c)+'0';
                    c=(c<0)? 1: 0;
                }
                while (p>0){
                    c=n[--p]-'0'-c;
                    t[--m]=((c<0)?c+10: c)+'0';
                    c=(c<0)? 1: 0;
                }
                while (q>0){
                    c='0'-d.n[--q]-c;
                    t[--m]=((c<0)?c+10: c)+'0';
                    c=(c<0)? 1: 0;
                }
                if(c>0) {t[--m]='-'; }
                else for(k=k-2; m<k && t[m]=='0'; m++);
                DEC x(t+m);
                delete t;
                return x;
            };
        };
}
DEC DEC::operator*(const DEC &d) const{
        if(n[0]=='0' ||d.n[0]=='0') return DEC("0");
        switch(n[0]){
        case '-':
            switch(d.n[0]){
            case '-':
                return (-d)*(-*this);
            default :
```

```
            return -(-*this*d);
        };
    default :
        switch(d.n[0]){
        case '-':
            return -(*this*(-d));
        default :
            int v,w, c, p=strlen(n), q=strlen(d.n);
            DEC r("0");
            DEC b((p>q)?n:d.n);
            DEC m((p>q)?d.n:n);
            for(v=0, w=((p>q)?q:p); v<w;    v++){
                char *t=new char[strlen(r.n)+2];
                r=strcat(strcpy(t, r.n), "0");
                delete t;
               for(c=m.n[v]-'0'; c>0; c--)      r=r+b;
            }
            if (strlen(r.n)==0) return DEC("0");
            return r;
        }
    }
}
DEC DEC::operator/(const DEC &d) const{
    if(d.n[0]=='0') throw new exception;
    switch(n[0]){
    case '-':
        switch(d.n[0]){
        case '-':
            return (-*this)/(-d);
        default :
            return -(-*this/d);
        };
    default :
        switch(d.n[0]){
        case '-':
            return -(*this/(-d));
        default :
            if(d>*this) return DEC("0");
            int v,w, c, p=strlen(n), q=strlen(d.n);
```

```
                char *r=new char[p+1], *t=new char[p+1];
                w=p-q+1;
                for(v=0; v<q; v++) t[v]=n[v]; t[q]=0;
                for(v=0; v<w;    v++){
                    c=0;
                    while (DEC(t)>d) {strcpy(t, (DEC(t)-d).n); c++; }
                    r[v]=c+'0';    r[v+1]=0;
                    if(t[0]=='0')    t[0]=n[q+v];
                    else{
                        c=strlen(t);
                        t[c]=n[q+v];
                        t[c+1]=0;
                    }
                }
                for(v=0; v<p && r[v]=='0'; v++);
                DEC s(r+v);
                delete r;
                delete t;
                return s;
            }
        }
}
DEC DEC::operator%(const DEC &d)const{ return *this-(*this/d)*d; }
DEC::operator const char *( ) const{ return n; }
DEC &DEC::operator++( ){ return *this=*this+"1"; }
DEC DEC::operator++(int){ DEC r(*this); ++(*this); return r; }
DEC &DEC::operator--( ){ return *this=*this-DEC("1"); }
DEC DEC::operator--(int){ DEC r(*this); --(*this); return r; }
void main( ){
    DEC w("1"), x("0"), y("1"), z("0");
    ++x="100";
    x="0";
    for(int k=1; k<5; k++) w=w*k;
    printf("%s\n", w);
    for(int k=1; k<5; k++) x=x+DEC(k);
    printf("%s\n", x);
    for(x=1;    x<DEC("10000000000000000000000000"); x++) y=y*x;
    printf("%s\n", y);
    for(x="1"; x<DEC("10000000000000000000000000"); x++) z=z+x;
```

```
        printf("%s\n", z);
    }
```

9.6 在下列程序中，main( )的语句都使用了哪些类型转换？它们都正确吗？

```
    struct Y;
    struct X {
        int i;
        X(int);
        operator int ( );
        X operator +(Y &);
    };
    struct Y{
        int i;
        Y(int);
        Y(X &);
        operator int ( );
    };
    void main(void) {
        int ret;
        X x=1;
        Y y=x;
        ret=y+10;
        ret=y+10*y;
        ret=x+y–1;
        ret=x*x+1;
    }
```

解：类型转换都是正确的，转换情况如下：

(1) ret=y+10，y 被转换为 int。

(2) ret=y+10*y，y 被转换为 int。

(3) ret=x+y–1，x+y 的结果为类型 X，该结果从 X 类型转换为 int。

(4) ret=x*x+1，x 从 X 类型转换为 int。

9.7 定义如下整数链表的函数成员。

```
    class LIST{
        struct NODE{
            int    value;
            NODE *next;
            NODE(int, NODE *);
        } *head;
    public:
        LIST(void);
```

```
    LIST(const LIST &);
    int find(int value)const;//查找元素 value，若找到则返回 1，否则返回 0
    NODE *operator [ ](int k)const;                  //取表的第 k 个元素
    virtual LIST operator +(const LIST &)const;  //表的合并运算
    virtual LIST &operator +(int value);             //插入一个元素
    virtual LIST &operator –(int value);             //删除一个元素
    virtual LIST &operator +=(const LIST &);     //表的合并运算
    virtual LIST &operator +=(int value);            //插入一个元素
    virtual LIST &operator –=(int value);            //删除一个元素
    virtual LIST &operator =(const LIST &);
    virtual ~LIST(void);
};
```

解：注意 NODE *operator [ ](int k)const 函数，operator [ ]通常用来做表、矩阵等的下标运算。函数成员定义如下：

```
LIST::LIST(void){head=0; }
LIST::LIST(const LIST &t){
    NODE *&p=head, *h=t.head;
    while(h!=0){
        p=new NODE(h->value, 0);
        h=h->next;
        p=p->next;
    }
}
int LIST::find(int value)const{
    NODE *h=head;
    while(h!=0 && h->value!=value){
        h=h->next;
    }
    return h!=0;
}
LIST::NODE* LIST::operator [ ](int k)const{
    NODE *h=head;
    for(int i=0; i<k; i++) if(h) h=h->next;
    return h;
}
LIST LIST::operator +(const LIST &t)const{
    LIST r(t);
    NODE *&p=r.head, *q=r.head, *h=head;
    while(h!=0){
```

```
            p=new NODE(h->value, 0);
            h=h->next;
            p=p->next;
        }
        p=q;
        return r;
}
LIST &LIST::operator +(int value){
        head=new NODE(value, head);
        return *this;
}
LIST &LIST::operator -(int value){
        NODE *&h=head, *p;
        while(h!=0 && h->value!=value) h=h->next;
        if(h){ p=h; h=h->next; delete p; }
        return *this;
}
LIST &LIST::operator +=(const LIST &t){ return *this=*this+t; }
LIST &LIST::operator +=(int value){ return *this=*this+value; }
LIST &LIST::operator -=(int value){ return *this=*this-value; }
LIST &LIST::operator =(const LIST &t){
        NODE *&p=head, *m=head, *n, *h=t.head;
        while(m!=0){
            n=m->next;
            delete m;
            m=n;
        }
        while(h!=0){
            p=new NODE(h->value, 0);
            h=h->next;
            p=p->next;
        }
        return *this;
}
LIST::~LIST(void){
        NODE *m=head, *n;
        while(m!=0){
            n=m->next;
            delete m;
```

```
            m=n;
        }
    }
```

9.8 定义如下集合类的函数成员。

```
class SET{
    int *elem;                                  //存放集合元素的动态内存
    int count;                                  //目前元素个数
   const int total;                             //最大元素个数
public:
    SET(int total);
    SET(const SET &);
    int find(int val)const;                     //找到值为 val 元素返回 1，否则返回 0
    int full(void)const;                        //集合满时返回 1，否则返回 0
    int empty(void)const;                       //集合空时返回 1，否则返回 0
    virtual SET operator +(const SET &)const;   //集合的并集
    virtual SET operator –(const SET &)const;   //集合的差集
    virtual SET operator *(const SET &)const;   //集合的交集
    virtual SET &operator <<(int value);        //增加一个元素
    virtual SET &operator >>(int value);        //删除一个元素
    virtual SET &operator + =(const SET &);     //集合的并集
    virtual SET &operator – =(const SET &);     //集合的差集
    virtual SET &operator * =(const SET &);     //集合的交集
    virtual SET &operator <<=(int value);       //增加一个元素
    virtual SET &operator >>=(int value);       //删除一个元素
    virtual SET &operator =(const SET &);
    virtual ~SET(void);
};
```

解：注意数据成员 total 的初始化，由于 total 是 const 成员，其初始化一定不能出现在构造函数的函数体内。函数成员定义如下:

```
#include <iostream.h>
SET::SET(int total):total((elem=new int[total])?total:0){ count=0; }
SET::SET(const SET &s):total((elem=new int[s.total])?s.total:0){
    int k;
    count=0;
    if(elem){ for(count=s.count, k=0; k<count; k++) elem[k]=s.elem[k]; }
}
int SET::find(int val)const{
    int k;
    for(k=0; k<count; k++) if(elem[k]==val) return 1;
```

```
    return 0;
}
int SET::full(void)const{ return count==total; }
int SET::empty(void)const{ return count==0; }
SET SET::operator +(const SET &s)const{
    int k;
    SET r(total+s.total);
    for(k=0; k<s.count; k++) r.elem[k]=s.elem[k];
    for(k=0; k<count; k++) r<<elem[k];
    return r;
}
SET SET::operator -(const SET &s)const{
    int k;
    SET r(*this);
    for(k=0; k<s.count; k++) r>>s.elem[k];
    return r;
}
SET SET::operator *(const SET &s)const{ return *this-(*this-s); }
SET &SET::operator <<(int value){
    if(find(value)) return *this;
    if(count<total)elem[count++]=value;
    return *this;
}
SET &SET::operator >>(int value){
    int k;
    for(k=0; k<count; k++)
        if(elem[k]==value){
            for(count--; k<count; k++) elem[k]=elem[k+1];
            return *this;
        }
    return *this;
}
SET &SET::operator +=(const SET &s){ return *this=*this+s; }
SET &SET::operator -=(const SET &s){ return *this=*this-s; }
SET &SET::operator *=(const SET &s){ return *this=*this*s; }
SET &SET::operator <<=(int value){ return *this=*this<<value; }
SET &SET::operator >>=(int value){ return *this=*this>>value; }
SET &SET::operator =(const SET &s){
    this->~SET( );                          //相当执行 delete    elem;
```

```
        new (this) SET(s);                    //利用当前对象壳子重新构造
        return *this;
    }
    SET::~SET(void){ if(elem) { delete elem, elem=0; }}
```

9.9 定义如下队列类的函数成员。

```
    class QUEUE{
        struct NODE{
            int    value;
            NODE *next;
            NODE(int v, NODE *n);
        } *head;
    public:
        QUEUE(void);
        QUEUE(const QUEUE &);
        virtual operator int(void)const;      //返回队列元素个数
        virtual int operator>>(int &);        //从队列中取出一个元素
        virtual int operator<<(int);          //往队列中加入一个元素
        virtual ~QUEUE(void);
    };
```

解：类似 cin 和 cout，用 operator<<和 operator>>表示加入和取出。函数定义如下：

```
    QUEUE::QUEUE(void){ head=0; }
    QUEUE::QUEUE(const QUEUE &q){
        NODE *&p=head, *h=q.head;
        if(q.head) { head=0; return; }
        while(h){
            p=new NODE(h->value, 0);
            h=h->next;
        }
    }
    QUEUE::operator int(void)const{
        int c=0;
        NODE *h=head;
        while(h){
            c++;
            h=h->next;
        }
        return c;
    }
    int QUEUE::operator>>(int &v){
```

```
        NODE *&h=head;
        if(!h) return 0;
        while(h->next)      h=h->next;
        v=h->value;
        delete h;
        h=0;
        return 1;
    }
    int QUEUE::operator<<(int v){
        NODE *h=head;
        if(!(h=new NODE(v, head))) return 0;
        head=h;
        return 1;
    }
    QUEUE::~QUEUE(void){
        NODE *q;
        while(head){
            q=head;
            head=head->next;
            delete q;
        }
    }
```

9.10定义一个存放单词的字典类，编写其中的函数：

```
    class DICTIONARY {
        char    ** words;
        int       pos;                                  //当前可以存放单词的空闲位置
        const    int max;                               //词典可以存放单词的最大个数
    public:
        DICTIONARY (int max);                           //max 为能存放的最大单词个数
        virtual ~ DICTIONARY ( );
        virtual int operator( ) (const char *w) const;  //查 w 在词典中的位置，未查到返回-1
        virtual DICTIONARY & operator<< (const char *w);    //若 w 不在词典则加入
        virtual const char * operator[ ] (int n) const;             //取出第 n 个单词
    };
```

解：注意，const 成员 max 的初始化不能出现在构造函数的函数体内，因函数体内变量值的变化应被看作修改或重新赋值。同时注意析构函数是如何释放内存资源的，因为 wrods 是逐个申请单词内存的，故要逐个释放申请的单词内存。同时注意在许多地方都用了 const char *类型，用来表示相应的变量或返回值所存放的字符串是不可修改的。成员函数定义如下：

```
#include <string.h>
DICTIONARY::DICTIONARY(int max):max((words=new char*[max])?max:0), pos(0){ }
DICTIONARY::~DICTIONARY( )
{
    if (words) { for(int i=0; i<pos; i++) delete words[i]; delete words; words=0; }
}
DICTIONARY &DICTIONARY::operator<<(const char *w)
{
    if ((*this)(w)||pos>=max||!(words[pos]=new char[strlen(w)+1])) return *this;
    strcpy(words[pos++], w);
    return *this;
}
int DICTIONARY::operator( )(const char *w)const
{
    for (int i=0; i<pos; i++)       if (!strcmp(words[i], w)) return 1;
    return 0;
};
const char*DICTIONARY::operator[](int n)const { return n<pos?words[n]:0; }
```

第 10 章　模　板

一．基本内容

函数模板、函数模板实例、函数模板覆盖、类模板、类模板实例、模板覆盖。

二．学习要点

函数模板是一种用来抽象描述大量计算过程类似的函数的机制。C ++的函数模板既可避免书写大量代码，又能利用编译程序提供的类型检查机制，保证程序在生成函数实例时保证语法正确性。声明模板的保留字为 template，声明中模板的参数表必须用尖括号括起来，每个参数必须在函数参数表中至少出现一次。在利用函数模板生成模板函数时，函数模板参数表的每个参数都只能用类型替换。

可以先声明模板原型，再定义函数模板。模板中的函数可以内联，也可以定义缺省参数和省略参数。但使用缺省参数在生成模板函数一般会产生类型转换问题。所以，最好不要使用缺省参数。

根据函数模板生成的函数实例称为模板函数或函数模板实例。当程序中出现函数调用指令时，编译程序可根据实参类型自动生成模板函数。在 ISO WG21/ANSI X3J16 标准中，可用 template 加上函数原型声明生成模板函数。

有时候，根据函数模板生成的模板函数没有实际意义，这种情况经常出现在函数参数为指针类型时。例如，有一个比较两个参数大小的函数模板 max，当用字符串指针代替其参数生成模板函数时，生成的模板函数仅比较字符串指针的大小，而不是比较字符串指针的所指向的存储单元的内容，这种比较对字符串运算来说没有实际意义。为此，必须定义字符串比较函数以覆盖函数模板生成的模板函数。

覆盖模板函数时一定要给出函数的完整定义。在只给出函数原型声明的情况下，根据函数模板能生成原型所定义的模板函数。对于通过原型声明生成的模板函数，在调用函数时函数的参数能自动进行类型转换。

在进行函数调用时，如果生成了模板函数并定义了覆盖函数，则编译程序采用如下策略确定调用函数：

① 首先寻找一个实参同形参完全匹配的覆盖函数，如果找到则调用该函数。

② 如果能通过函数模板生成实例函数，并且参数匹配则调用该函数。

③ 通过强制类型转换，寻找能够与实参匹配的覆盖函数，如果找到则调用该函数。

④ 如果所有努力失败，则给出错误信息。

类模板也称为类属类或参数化的类，用来抽象描述大量类型相似的类。编译程序将根据类型参数生成相应的类实例，也称为模板类或类模板实例。利用类模板可以显著减少程序书写量，但不会减少程序编译后执行代码的长度。

在类模板中，除了可以使用类型参数外，还可以使用确定类型变量作参数。类模板的成员函数可以内联，也可以定义缺省参数和省略参数。如果缺省参数使用的确定类型变量

作参数，则在生成模板类时一般不会产生问题。

也可以在类模板声明友元时定义函数模板，此时，友元函数模板同类模板建立了关联，即一旦生成实例类便会生成实例函数，或者一旦生成实例函数便生成实例类。普通函数模板和函数成员模板均可定义为类模板的友元模板。

在某些情况如类型参数为指针类型的下，编译程序根据类模板生成的模板类也许不太理想，此时便需要直接提供模板类定义来覆盖编译程序生成的模板类。

三．习题答案

10.1设计一个 abs 函数模板，用于计算整数及浮点数的绝对值。

解：函数模板 abs 如下：

```
template <class T>
T abs(T x)
{
    if(x<0) x=-x;
    return x;
}
```

10.2设计一个 operator ==函数模板，用于比较各类型的数据相等与否。

解：函数模板 operator ==如下：

```
template <class T>
int operator ==(const T &x, const T &y)
{
    return x==y;
}
```

10.3设计一个 STACK 类型的模板，用于实例化各种类型的栈。

解：STACK 类型的模板如下：

```
template <class T=int>
class STACK
{
    T     *stk;
    int    top;
    const    int    tot;
public:
    STACK(int);
    ~STACK( );
    T &operator[ ](int);
    int operator>>(T&);
    int operator<<(const T&);
};
template <class T>
```

```
STACK <T>::STACK(int t):tot((stk=new T[t])?t:0), top(0){ }
template <class T>
STACK <T>::~STACK( ){ if(stk){ delete stk; stk=0; } }
template <class T>
T& STACK <T>::operator[ ](int x){ return stk[x]; }
template <class T>
int STACK <T>::operator>>(T&p){
    if(top>0){ p=stk[--top]; return 1; }
    return 0;
}
template <class T>
int STACK <T>::operator<<(const T &p){
    if(top==tot) return 0;
    stk[top++]=p;
    return 1;
}
```

10.4设计一个 QUEUE 类型的模板，用于实例化各种类型的队列。

解：参见习题 3.4 关于 volatile 的用法，可知为什么没有将成员函数 operator int( )const 定义为 operator int( )const volatile。QUEUE 类型的模板如下：

```
template <class T=int>
class QUEUE
{
    T     *q;
    int    f, r;
    const    int s;
public:
    QUEUE(int);
    ~QUEUE( );
    operator int( )const;                  //得到队列中的元素个数
    int operator>>(T&)volatile;            //从队列中的取出一个元素
    int operator<<(const T&)volatile;    //将元素加到队列首部
};
template <class T>
QUEUE <T>::QUEUE(int s):s((q=new T[s])?s:0), f(0), r(0){ }
template <class T>
QUEUE <T>::~QUEUE( ){ if(q){ delete q; q=0; } }
template <class T>
int QUEUE <T>::operator>>(T&p)volatile{
    if(s==0)    return 0;
```

```
    if(f==r)    return 0;
    p=q[f=(f+1)%s];
    return 1;
}
template <class T>
int QUEUE <T>::operator<<(const T&p)volatile{
    if(s==0)    return 0;
    int n=(r+1)%s;
    if(n==f) return 0;
    q[r=n]=p;
    return 1;
}
template <class T>
QUEUE <T>::operator int ( )const {
    if(s==0)    return 0;
    return(r-f+s)%s;
}
```

10.5设计一个 LIST 类型的模板，用于实例化各种类型的线性表。

解：LIST 类型的模板如下：

```
template <class T=int>
class LIST{
    struct NODE{
        T       value;
        NODE *next;
        NODE(T, NODE *);
    } *head;
public:
    LIST(void);
    LIST(const LIST &);
    int find(T value)const;//查找元素 value，若找到则返回 1，否则返回 0
    NODE * operator [ ](int k)const;                //取表的第 k 个元素
    virtual LIST operator +(const LIST &)const;  //表的合并运算
    virtual LIST &operator +(T value);              //插入一个元素
    virtual LIST &operator -(T value);              //删除一个元素
    virtual LIST &operator +=(const LIST &);    //表的合并运算
    virtual LIST &operator +=(T value);             //插入一个元素
    virtual LIST &operator -=(T value);             //删除一个元素
    virtual LIST &operator =(const LIST &);
    virtual ~LIST(void);
```

```
};
template <class T>
LIST<T>::LIST(void){head=0; }
template <class T>
LIST<T>::LIST(const LIST &t){
    NODE *&p=head, *h=t.head;
    while(h!=0){
        p=new NODE(h->value, 0);
        h=h->next;
        p=p->next;
    }
}
template <class T>
int LIST<T>::find(T value)const{
    NODE *h=head;
    while(h!=0 && h->value!=value){
        h=h->next;
    }
    return h!=0;
}
template <class T>
LIST<T>::NODE* LIST<T>::operator [ ](int k)const{
    NODE *h=head;
    for(int i=0; i<k; i++) if(h) h=h->next;
    return h;
}
template <class T>
LIST<T> LIST<T>::operator +(const LIST &t)const{
    LIST r(t);
    NODE *&p=r.head, *q=r.head, *h=head;
    while(h!=0){
        p=new NODE(h->value, 0);
        h=h->next;
        p=p->next;
    }
    p=q;
    return r;
}
template <class T>
```

```
LIST<T> &LIST<T>::operator +(T value){
    head=new NODE(value, head);
    return *this;
}
template <class T>
LIST<T> &LIST<T>::operator -(T value){
    NODE *&h=head, *p;
    while(h!=0 && h->value!=value) h=h->next;
    if(h){ p=h; h=h->next; delete p; }
    return *this;
}
template <class T>
LIST<T> &LIST<T>::operator +=(const LIST &t){ return *this=*this+t; }
template <class T>
LIST<T> &LIST<T>::operator +=(T value){ return *this=*this+value; }
template <class T>
LIST<T> &LIST<T>::operator -=(T value){ return *this=*this-value; }
template <class T>
LIST<T> &LIST<T>::operator =(const LIST &t){
    NODE *&p=head, *m=head, *n, *h=t.head;
    while(m!=0){
        n=m->next;
        delete m;
        m=n;
    }
    while(h!=0){
        p=new NODE(h->value, 0);
        h=h->next;
        p=p->next;
    }
    return *this;
}
template <class T>
LIST<T>::~LIST(void){
    NODE *m=head, *n;
    while(m!=0){
        n=m->next;
        delete m;
        m=n;
```

```
        }
    }
```

10.6设计一个二维 ARRAY 类型的模板，用于实例化各种类型的二维数组。

解：本习题用了异常处理，也可以用其他方法检查下标。二维 ARRAY 类型的模板如下：

```
#include <stdio.h>
#include <exception.h>
template <class T>
class ARRAY{
    T       *a;
    int       r,c;
public:
    ARRAY(int x, int y);
    ARRAY(const ARRAY&);
    ~ARRAY( );
    T &operator( )(int x, int y);
    ARRAY &operator=(const ARRAY &);
    ARRAY operator+(const ARRAY &)const;
    ARRAY operator-(const ARRAY &)const;
    ARRAY operator*(const ARRAY &)const;
    ARRAY operator!( )const;
};
template <class T>
ARRAY<T>::ARRAY(int x, int y){
    if(a=new T[x*y]){ r=x; c=y; }
}
template <class T>
ARRAY<T>::ARRAY(const ARRAY<T>&m){
    if(a=new T[m.r*m.c]){ r=m.r; c=m.c; }
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            a[h*c+k]=m.a[h*c+k];
}
template <class T>
ARRAY<T>::~ARRAY( ){ if(a){ delete a; a=0; r=c=0; } }
template <class T>
T &ARRAY<T>::operator( )(int x, int y){ return a[x*c+y]; }
template <class T>
ARRAY<T> &ARRAY<T>::operator=(const ARRAY<T> &m){
    delete a;
```

```
    if(a=new T[m.r*m.c]){ r=m.r; c=m.c; }
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            a[h*c+k]=m.a[h*c+k];
    return *this;
}
template <class T>
ARRAY<T> ARRAY<T>::operator+(const ARRAY<T> &m)const{
    if(r!=m.r||c!=m.c) throw new exception;
    ARRAY<T> v(m);
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            v.a[h*c+k]+=a[h*c+k];
    return v;
}
template <class T>
ARRAY<T> ARRAY<T>::operator-(const ARRAY<T> &m)const{
    if(r!=m.r||c!=m.c) throw new exception;
    ARRAY<T> v(*this);
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            v.a[h*c+k]-=m.a[h*c+k];
    return v;
}
template <class T>
ARRAY<T> ARRAY<T>::operator*(const ARRAY<T> &m)const{
    if(c!=m.r) throw new exception;
    ARRAY<T> v(r, m.c);
    for(int h=0; h<r; h++)
        for(int k=0; k<m.c; k++)
            for(int p=0; p<c; p++)
                v.a[h*m.c+k]+=a[h*c+p]*m.a[p*m.c+k];
    return v;
}
template <class T>
ARRAY<T> ARRAY<T>::operator!( )const{
    ARRAY<T> v(c, r);
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
```

```
                v.a[h*r+k]-=a[h*c+k];
        return v;
}
void main( ){
        ARRAY<int>    a(10, 20), b(20,30), c(10,20), d(1,1);
        a(1,2)=3;    b(2,4)=6;    c(1,2)=10;
        d=a*b;
        d=a+c;
        d=!a;
}
```

第 11 章　异常处理与断言

一．基本内容

异常、引发异常、捕获异常、异常接口、异常类型、异常对象析构、断言。

二．学习要点

异常是一种意外中断程序正常处理流程的事件。C 和 C++均提供了异常处理功能。C 只能处理 unsigned 类型的异常，C++则可以处理各种类型的异常。C++提供了面向对象的异常处理功能，因此，本章只准备介绍 C++的面向对象的异常处理机制。

异常既可以被硬件引发，又可以被软件引发。由硬件引发的异常通常由中断服务进程产生，例如，算术运算溢出和除数为 0 所产生的异常；由软件引发的异常由 throw 语句产生，操作系统和应用程序都可能引发异常。

当程序引发一个异常时，在引发点建立一个描述异常的对象，之后控制被转移到该异常的处理过程，在引发点建立的对象用于初始化异常处理过程的参数。在引发异常之前，可以先声明一个类型，用于描述异常发生时的现场及错误信息，以便异常处理过程在捕获和处理异常获得足够的信息。

引发异常通常称为 throw，而捕获异常通常称为 catch。引发异常后，首先在引发异常的函数内部寻找异常处理过程，如果没有找到异常处理过程，则在调用该函数的函数内继续寻找，就这样一直找到顶层调用函数。如果顶层调用函数也没有处理异常，则由 C++的监控系统处理异常，监控系统通常会就此终止程序。异常处理过程在处理完异常后，可以通过没有参数的 throw 继续传播异常，还可以通过带参数的 throw 引发新的异常。

异常捕获过程必须定义参数，指明要捕获的异常对象类型，当异常对象类型同参数类型相容时才能被捕获。注意父类指针类型可以和子类指针类型相容，而省略参数则可以和任何类型相容。在安排异常捕获过程时，通常将捕获子类指针类型的过程置于捕获父类指针类型的过程之前，将省略参数的异常捕获过程放在所有异常捕获过程的最后，否则，这些异常捕获过程便没有机会捕获异常。

为了使其他人能方便地了解某个函数可能引发的异常，可在声明该函数时列出该函数可能引发的所有异常，通过异常接口声明的异常都是由该函数引发的、而其自身又不想捕获和处理的异常。异常接口定义的异常出现在函数的参数表后面，用 throw 带参数表列出要引发的异常类型。参数表为空表示不引发任何异常，没有 throw 则可以引发任何类型的异常。

一个不引发任何异常的函数引发的异常或者引发的同 throw 预先定义不符的异常称为不可意料的异常。不可意料的异常通常不是由程序引发的，可由不可意料的异常处理过程 unexpected 处理，unexpected 一般会终止应用程序的执行。unexpected 可以引发一个已经声明了的异常。通过 set_unexpected 过程，可以将不可意料的异常处理过程设置为程序自定义的异常处理过程。

C++的新版编译程序允许函数模板和模板函数定义异常接口，并且允许类模板及模板类

的函数成员定义异常接口。

异常类型可以是任何类型，C++提供了一个标准的异常类型 exception，可作为定义异常类型的基类。exception 的函数成员不再引发任何异常，其中，函数成员 what 返回一个只读字符串，一般必须在派生类中重新定义函数成员 what。

C++为标准类库提供了各种各样的异常处理类。应用程序可以使用这些类库和相应的异常处理，也可以以上述异常类为基础派生出新的异常类。C++标准类库的异常处理涉及文档处理异常、内存管理异常、资源管理异常、网络操作异常、数据库操作异常、ODBC 连接异常、动态连接异常等各种类型，使用时可查阅相关类库的联机资料。

异常处理也可能产生内存泄漏问题，而且这种问题常被人们忽视。程序在引发异常时产生一个异常对象，在这个对象作为实参传递给异常处理过程时也存在浅拷贝构造问题。因此，如果异常对象含有指向动态内存的指针成员，则异常对象所属的类必须定义深拷贝构造函数。

如果在执行构造函数的过程中引发了异常，则只有构造完毕的基类对象得到析构，没有完全构造好的派生类对象将不会析构。同理，如果派生类有虚基类和对象成员，则构造好的虚基类和对象成员将被析构。一个局部数组如果只构造好一部分元素，则只有一部分元素被析构。对于未完全构造好因而不能自动析构的对象，必须在引发异常之前作好释放内存等善后处理。

断言 assert 用于在程序的检查点设置约束条件，使用时必须包含 assert.h 头文件。如果断言为真，则程序继续正常执行；否则，输出断言表达式、断言所在的程序文件名称以及断言所在行的行号，然后调用 abort 终止程序执行。

三．习题答案

11.1 什么叫异常？怎样在程序中引发异常？怎样捕获异常？

解：（1）异常是一种意外中断程序正常处理流程的事件。（2）异常既可以被硬件引发，又可以被软件引发。由硬件引发的异常通常由中断服务进程产生，由软件引发的异常由 throw 语句产生，操作系统和应用程序都可能引发异常。（3）首先在引发异常的函数内部寻找异常处理过程，如果没有找到异常处理过程，则在调用该函数的函数内继续寻找，就这样一直找到最外层调用函数。如果最外层调用函数也没有处理异常，则程序的监控系统通常会终止程序。

11.2为 STRING 类定义异常处理过程，以便堆空间不够时处理异常。

```
#include <stdio.h>
#include <exception.h>
class    MEMEXCEPTION: public exception{
public:
        const char*what( ) const throw( ){ return "Memory not enough!";}
};
class    POPEXCEPTION: public exception{
public:
        const char*what( ) const throw( ){ return "Pop from empty stack!";}
```

```
};
class    NODE{
        int      value;
        NODE *next;
public:
        NODE(int value, NODE *next);
        int getv( ){ return value;};
        NODE *getn( ){return next;};
};
NODE::NODE(int v, NODE* n)
{
        value=v;
        next=n;
}
class    STACK{
        NODE *head;
public:
        STACK( ){ head=0; };
        STACK& operator<<(v);
        STACK& operator>>(int &v);
        ~STACK( );
};
STACK& STACK::operator<<(int v)
{
        NODE *p=new NODE(v, head);
        if(p==0) throw MEMEXCEPTION( );
        head=p;
        return *this;
}
STACK& STACK::operator>>(int &v )
{
        NODE *p=head;
        if(head==0) throw POPEXCEPTION( );
        v=head->getv( );
        head=head->getn( );
        delete p;
        return *this;
}
STACK::~STACK( )
```

```
{
    NODE *q, *p=head;
    while(p!=0){
        q=p->getn( );
        delete p;
        p=q;
    }
}
void main( ){
    STACK   stk;
    int          x,y,z;
    try{
        stk<<1<<2<<3<<4<<5<<6;
        stk>>x>>y>>z;
    }catch(POPEXCEPTION   p){
        printf(p.what( ));
    }catch(MEMEXCEPTION   m){
        printf(m.what( ));
    }
}
```

11.3为二维整数数组 ARRAY 定义异常处理过程，以处理下标越界和堆空间不够等异常。

解：二维整数数组 ARRAY 如下：

```
#include <stdio.h>
#include <exception.h>
class   MEMEXCEPTION: public exception{
public:
    const char*what( ) const throw( ){ return "Memory not enough!";}
};
class   DIFEXCEPTION: public exception{
public:
    const char*what( ) const throw( ){ return "added matrix differ in size!";}
};
class   MATEXCEPTION: public exception{
public:
    const char*what( ) const throw( ){ return "multiplied matrix not match!";}
};
class ARRAY{
    int     *a;
    int      r,c;
```

```
public:
    ARRAY(int x, int y);
    ARRAY(const ARRAY&);
    ~ARRAY( );
    int &operator( )(int x, int y);
    ARRAY &operator=(const ARRAY &);
    ARRAY operator+(const ARRAY &)const;
    ARRAY operator-(const ARRAY &)const;
    ARRAY operator*(const ARRAY &)const;
    ARRAY operator!( )const;
};
ARRAY::ARRAY(int x, int y){
    if(!(a=new int[x*y])) throw MEMEXCEPTION( );
    r=x; c=y;
}
ARRAY::ARRAY(const ARRAY&m){
    if(!(a=new int[m.r*m.c])) throw MEMEXCEPTION( );
    r=m.r; c=m.c;
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            a[h*c+k]=m.a[h*c+k];
}
ARRAY::~ARRAY( ){ if(a){ delete a; a=0; r=c=0; } }
int &ARRAY::operator( )(int x, int y){ return a[x*c+y]; }
ARRAY &ARRAY::operator=(const ARRAY &m){
    delete a;
    if(!(a=new int[m.r*m.c])) throw MEMEXCEPTION( );
    r=m.r; c=m.c;
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            a[h*c+k]=m.a[h*c+k];
    return *this;
}
ARRAY ARRAY::operator+(const ARRAY &m)const{
    if(r!=m.r||c!=m.c) throw DIFEXCEPTION( );
    ARRAY v(m);
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            v.a[h*c+k]+=a[h*c+k];
```

```
    return v;
}
ARRAY ARRAY::operator-(const ARRAY &m)const{
    if(r!=m.r||c!=m.c) throw DIFEXCEPTION( );
    ARRAY v(*this);
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            v.a[h*c+k]-=m.a[h*c+k];
    return v;
}
ARRAY ARRAY::operator*(const ARRAY &m)const{
    if(c!=m.r) throw MATEXCEPTION( );
    ARRAY v(r, m.c);
    for(int h=0; h<r; h++)
        for(int k=0; k<m.c; k++)
            for(int p=0; p<c; p++)
                v.a[h*m.c+k]+=a[h*c+p]*m.a[p*m.c+k];
    return v;
}
ARRAY ARRAY::operator!( )const{
    ARRAY v(c, r);
    for(int h=0; h<r; h++)
        for(int k=0; k<c; k++)
            v.a[h*r+k]-=a[h*c+k];
    return v;
}
void main( ){
    try{
        ARRAY   a(10, 20), b(20,30), c(10,20), d(1,1);
        a(1,2)=3; b(2,4)=6; c(1,2)=10;
        d=a*b;
        d=a+c;
        d=!a;
    }catch(DIFEXCEPTION   d){
        printf(d.what( ));
    }catch(MATEXCEPTION   m){
        printf(m.what( ));
    }catch(MEMEXCEPTION   m){
        printf(m.what( ));
```

```
        }
    }
```

11.4为 QUEUE 类增加异常处理过程，以便在堆空间不够时处理异常。

解：QUEUE 类定义如下：

```
    #include <stdio.h>
    #include <exception.h>
    class    MEMEXCEPTION: public exception{
    public:
         const char*what( ) const throw( ){ return "Memory not enough!";}
    };
    class    FULLEXCEPTION: public exception{
    public:
         const char*what( ) const throw( ){ return "Queue is full!";}
    };
    class    EMPTYEXCEPTION: public exception{
    public:
         const char*what( ) const throw( ){ return "Queue is empty!";}
    };
    class QUEUE{
        char *queue;
        int      size, front, rear;
    public:
        QUEUE& operator<<(char elem);
        QUEUE& operator>>(char &elem);
        QUEUE(int size);
        ~QUEUE(void);
    };
    QUEUE::QUEUE(int sz)
    {
        if(!(queue=new char[size=sz])) throw MEMEXCEPTION( );
        front=rear=0;
    }
    QUEUE::~QUEUE(void)
    {
        if(queue){
            delete queue;
            queue=0;
            front=0;
            rear=0;
```

```
        }
    }
    QUEUE& QUEUE::operator<<(char elem)
    {
        if((rear+1)%size==front) throw FULLEXCEPTION( );
        queue[rear=(rear+1)%size]=elem;
        return *this;
    }
    QUEUE& QUEUE::operator>>(char &elem)
    {
        if(rear==front) throw EMPTYEXCEPTION( );
        elem=queue[front=(front+1)%size];
        return *this;
    }
    void main(void)
    {
        try{
            int     a,b,c,d,e;
            QUEUE queue(20);
            queue<<1<<2<<3<<4<<5;
            queue>>a>>b>>c>>d>>e;
        }catch(FULLEXCEPTION    f){
            printf(f.what( ));
        }catch(EMPTYEXCEPTION    e){
            printf(e.what( ));
        }catch(MEMEXCEPTION    m){
            printf(m.what( ));
        }
    }
```

11.5 为复数类 COMPLEX 定义异常处理过程，以便在除以零时处理异常。

解：复数类定义 COMPLEX 如下：

```
    #include <stdio.h>
    #include <exception.h>
    class    DIVEXCEPTION: public exception{
    public:
         const char*what( ) const throw( ){ return "Divided by zero!";}
    };
    class    COMPLEX{                        //real, imag 分别为复数的实部和虚部，
        double      real, imag;              //若 x=3+5i，则 x.real=3，x.imag=5
```

```
public:
    COMPLEX(double r=0);
    COMPLEX(double r, double i);
    COMPLEX operator - ( ) const;      //将实部和虚部的符号求反
    COMPLEX operator + (const COMPLEX &c) const;
    COMPLEX operator - (const COMPLEX &c) const;
    COMPLEX operator * (const COMPLEX &c) const;
    COMPLEX operator / (const COMPLEX &c) const;
};
COMPLEX::COMPLEX(double r){    real=r;    imag=0;    }
COMPLEX::COMPLEX(double r, double i){    real=r;    imag=i;    }
COMPLEX COMPLEX::operator - ( ) const{    return COMPLEX(-real, -imag); }
COMPLEX COMPLEX::operator + (const COMPLEX &c) const
{    return COMPLEX(real+c.real, imag+c.imag); }
COMPLEX COMPLEX::operator - (const COMPLEX &c) const
{    return COMPLEX(real-c.real, imag-c.imag); }
COMPLEX COMPLEX::operator * (const COMPLEX &c) const
{    return COMPLEX(real*c.real-imag*c.imag, imag*c.real+c.imag*real); }
COMPLEX COMPLEX::operator / (const COMPLEX &c) const{
    double d=c.real*c.real+c.imag*c.imag;
    if(d==0) throw DIVEXCEPTION( );
    return COMPLEX((real*c.real+imag*c.imag)/d, (imag*c.real-c.imag*real)/d);
}
void main(void){
    try{
        COMPLEX    a,b(4), c(6,7), d;
        d=-a;
        d=a-b/c;
    }catch(DIVEXCEPTION    m){
        printf(m.what( ));
    }
}
```

11.6 在对象没有完全构造时出现异常，应当怎样处理？

解：如果在执行构造函数的过程中引发了异常，则只有构造完毕的基类对象得到析构，没有完全构造好的派生类对象将不会析构。同理，如果派生类有虚基类和对象成员，则构造好的虚基类和对象成员将被析构。一个局部数组如果只构造好一部分元素，则只有一部分元素被析构。对于未完全构造好因而不能自动析构的对象，必须在引发异常之前作好释放内存等善后处理。

面向对象程序设计模拟试卷一

一. 选择题(15)。

1. 关于构造的叙述_____正确:
   A. 最先构造虚基类　　B. 最先构造基类
   C. 最先构造派生类的对象成员　　D. 都不对
2. 关于静态数据成员的叙述_____正确:
   A. 公有的可在类体外初始化　　B. 私有的不能在类体外初始化
   C. 私有和保护的不能在类体外初始化　　D. 都可以且必须在体外初始化
3. 若派生类函数不是基类的友元，关于该函数访问基类成员_____正确:
   A. 公有的可被派生类函数访问　　B. 都可以被派生类函数访问
   C. 公有和保护的可被派生类函数访问　D. 都不对
4. 关于函数的所有缺省参数的叙述_____正确:
   A. 只能出现在参数表的最左边　　B. 只能出现在参数表的最右边
   C. 必须用非缺省的参数隔开　　D. 都不对
5. 使用 friend、virtual、static 说明函数的叙述_____正确:
   A. 必须同时使用三个　　B. 只能同时用其中两个
   C. 只能独立单个地使用　　D. 都不对

二. 指出各类的成员及其存取属性(20) 。

```
class A{
      int   a;
protected:
      int   b;
public:
      int   c;
};
class B: protected A{
      int   d;
protected:
      int   e;
public:
     A::c;
      int   f;
};
```

```
class C:   A{
      int   g;
protected:
      int   h;
public:
      int   i;
};
class D: B, C{
      int   j;
protected:
     B::b;
      int   k;
private:
      int   n;
};
```

三. 指出 main 中每行的输出结果(20) 。

```
#include <iostream.h>
struct A{A( ){ cout<<'A';}};
struct B{B( ){ cout<<'B';}};
struct C: A{C( ){ cout<<'C';}};
```

```
struct D: virtual B, C{D( ){ cout<<'D';}};
struct E: A{
      C    c;
      E( ): c( ){ cout<<'E';}
};
struct F: virtual B, C, D, E{
     F( ){ cout<<'F';}
};
void main( ){
      A    a;    cout<<'\n';
      B    b;    cout<<'\n';
      C    c;    cout<<'\n';
      D    d;    cout<<'\n';
      E    e;    cout<<'\n';
      F    f;    cout<<'\n';
}
```

四. 指出以下程序的语法错误及其原因(15) 。

```
class A{
            static int    a=0;
      protected:
            int    b;
      public:
            int    c;
            A(int);
            operator int( );
      } a(1, 2);
      class B: A{
            B(int);
            virtual int d;
            int    e;
      public:
            A::b;
            friend int operator =(B);
            static B(int, int);
} b=5;
class C: B{
public:
      int operator++(double);
};
```

```
int main( ){
      int    *A::*p, i;
       i=a.a;
       i=A(4);
       i=b.c;
       p=&A::c;
       i=b;
       return ;
}
```

五. 指出 main 变量 i 在每条赋值语句执行后的值(15) 。

```
int      x=2,    y=x+30;
struct A{
       static int    x;
       int    y;
public:
       operator int( ){ return x-y; }
       A operator ++(int){ return A(x++, y++); }
       A(int x=::x+2, int y=::y+3){ A::x=x;    A::y=y; }
       int &h(int &x);
};
int &A::h(int &x)
{
     for(int y=1; y!=1|| x<201; x+=11, y++)    if(x>200) { x-=21; y-=2;}
       return x-=10;
}
int A::x=23;
void main( ){
       A    a(54, 3),    b(65),    c;
       int    i,    &z=i,    A::*p=&A::y;
       z=b.x;
       i=a.x;
       i=c.*p;
       i=a++;
       i=::x+c.y;
       i=a+b;
       b.h(i)=7;
}
```

六. 为了没有误差地表示分数，定义类 FRACTION 来表示分数，请编程实现分数类中的除 cmd 外的所有函数(15)。

```
class FRACTION{ //对于6/7，numerator 存分子 6，denominator 存分母 7
    int     numerator, denominator;
    static int cmd(int x, int y);                          //求整数 x,y 的最大公约数
public:
    int operator>(const FRACTION&)const;                   //大于比较，例 6/7>2/3
    FRACTION(int num, int den=1);                          //num、den 各为分子和分母
    FRACTION operator*( )const;                            //分数约简，*30/36=5/6
    FRACTION operator+(const FRACTION&)const; //加法，  6/7+2/3=32/21
    FRACTION operator*(const FRACTION&)const; //乘法，  6/7*2/3=12/21=4/7
};
int   FRACTION::cmd(int x, int y){
    int r;
    if(x<y){   r=x;   x=y;   y=r;   }
    while(y!=0){   y=x%(r=y);   x=r; }
    return x;
}
```

面向对象程序设计模拟试卷二

一．单项选择题 （从下列各题四个备选答案选出一个正确答案，并将其代号填在题干前的括号内。答案选错或未作选择者，该题不得分。每小题 1 分，共 10 分）

（　）1．以下有关析构函数的叙述，选择正确的填入括号内。

A. 可以进行重载可以定义为虚函数

B. 不能进行重载可以定义为虚函数

C. 可以进行重载不能定义为虚函数

D. 不能进行重载不能定义为虚函数

（　）2．以下有关函数缺省参数的出现位置，选择正确的叙述填入括号内。

A. 必须全部出现在函数参数表的左部

B. 必须全部出现在函数参数表的右部

C. 必须全部出现在函数参数表的中间

D. 都不对

（　）3．关于两个同名函数重载的叙述，选择最为正确的填入括号内。

A. 两个函数的参数个数不同

B. 两个函数对应的参数类型不同

C. 两个函数的参数个数不同或对应的参数类型不同

D. 都不对

（　）4．关于 inline 保留字的用途，选择正确的叙述填入括号内。

A. 只能用于定义成员函数

B. 只能用于定义非成员函数

C. 可以定义成员函数及非成员函数

D. 都不对

（　）5．关于类的构造函数的定义位置，将最为正确的叙述填入括号内。

A. 只能在 private 下定义

B. 只能在 protected 下定义

C. 只能在 public 下定义

D. 定义位置没有限制

（　）6．如下修饰类体中函数的返回类型，将正确的用法填入括号内。

A. 同时使用 static 和 friend

B. 同时使用 static 和 virtual

C. 同时使用 friend 和 virtual

D. 不同时使用上述三个保留字中的任意两个

（　）7．在如下关于继承的叙述中，选择正确的填入括号内。

A. 私有继承用于实现类之间的 ISA 关系

B. 保护继承用于实现类之间的 ISA 关系

C. 公有继承用于实现类之间的 ISA 关系

D. 都不对

（　）8．对于用 union 定义的类 A，选择正确的叙述填入括号内。

A. 类 A 可以作为某个类的基类

B. 类 A 可以作为某个类的派生类

C. 类 A 既不能作基类又不能作派生类

D. 类 A 既可以作基类又可以作派生类

（　）9．有关在类中声明友元函数的叙述，将正确的答案填入括号内。

A. 只能在 private 下声明

B. 只能在 protected 下声明

C. 只能在 public 下声明

D. 声明位置没有限制

（　）10．在构造派生类 A 的对象时，选择正确的叙述填入括号内。

A. 最先构造派生类 A 的基类

B. 最先构造派生类 A 的虚基类

C. 最先构造派生类 A 的数据成员对象

D. 都不对

二．多项选择题（从下列各题四个备选答案选出二至四个正确答案，并将其代号填在题干前的括号内。答案选错或未选全者，该题不得分。每小题 2 分，共 10 分）

（　）11．将符号为分隔符的编号填入括号内。

A. m　　B. +　　C. {　　D. =

（　）12．将用于定义类的保留字的编号填入括号内。

A. class　　B. struct　　C. int　　D. union

（　）13．将用于释放 p=new int[8]的内存的编号填入括号内。

A. free (p)　　B. delete p　　C. delete []p　　D. sizeof　p

（　）14．将派生类成员函数可访问的基类成员编号填入括号内。

A. private　　B. protected　　C. public　　D. 所有成员

（　）15．将不能重载的运算符函数的编号填入括号内。

A. ? :　　B. ::　　C. %　　D. <<

三．填空题（阅读以下程序并填空。每题 1 分，共 10 分）

```
class A{
    int  a;
protected:
    int  b;
public:
    int  c;
}a;
```

```
class C:  A{
    int  g;
protected:
    int  h;
public:
    int  c;
}c;
```

```
class B: protected    A{                    struct D: B, C{
      int    d;                                  int    j;
protected:                                  protected:
              int    e;                              B::b;
   public:                                  }d;
      A::c;                                 void main(void){
   int    f;                                int    x;
   }b;                                           }
```

16．对于 main，对象 a 的私有成员为_____。
17．对于 main，对象 a 的保护成员为_____。
18．对于 main，对象 a 的公有成员为_____。
19．对于 main，对象 b 的私有成员为_____。
20．对于 main，对象 b 的保护成员为_____。
21．对于 main，对象 c 的保护成员为_____。
22．对于 main，对象 c 的公有成员为_____。
23．对于 main，对象 d 的保护成员为_____。
24．对于 main，对象 d 的公有成员为_____。
25．main 应使用语句 x=_____取对象 d 的基类 C 的成员 c 的值。

四．名词解释（每小题 3 分，共 15 分）

26．标识符：

27．重载：

28．异常：

29．类型强制：

30．抽象类：

五．判断改错题（阅读以下程序并判断各题是否正确，对正确的就在其题号前打 √；错误的在其题号前打 x，并在题干后的括号内更正。每小题 3 分，共 15 分）

```
#include <iostream.h>
struct A{A( ){ cout<<'A';}};
struct B{B( ){ cout<<'B';}};
struct C: A{C( ){ cout<<'C';}};
   struct D: virtual B, C{D( ){ cout<<'D';}};
   struct E: A {
   C    c;
```

```
E( ): c( ){ cout<<'E';}
};
struct F: virtual B, C, D, E{
F( ){ cout<<'F';}
};
void main( ){ /*……*/ }
```

(　　) 31．如 main 定义 A　a，则输出为 A。　　（_____）

(　　) 32．如 main 定义 C　c，则输出为 BAC。　（_____）

(　　) 33．如 main 定义 D　d，则输出为 BACD。　　（_____）

(　　) 34．如 main 定义 E　e，则输出为 AACE。　　（_____）

(　　) 35．如 main 定义 F　f，则输出为 BCDEF。　　（_____）

六．计算题（阅读如下程序并计算执行结果。共 20 分）

```
int   x=2,   y=x+30;
struct A{
      static int   x;
      int   y;
public:
      operator int( ){ return x+y; }
      A operator ++(int){ return A(x++, y++); }
      A(int x=::x+2, int y=::y+3){ A::x=x;   A::y=y; }
      int &h(int &x);
};
int &A::h(int &x)
{
      for(int y=1; y!=1 || x<201; x+=11, y++)
            if(x>200) { x-=21; y-=2;}
      return x-=10;
}
int A::x=23;
void main( ){
A   a(54, 3),   b(65),   c;
int   i,   &z=i,   A::*p=&A::y;
      i=b.x;                  //……①
      z=a.x;                  //……②
      i=c.*p;                 //……③
      i=a++;                  //……④
      i=::x+c.y;              //……⑤
      i=a+b;                  //……⑥
```

```
    b.h(i)=7;              //……⑦
}
```

36．语句①执行结束后 i=　　。（2 分）

37．语句②执行结束后 i=　　。（3 分）

38．语句③执行结束后 i=　　。（3 分）

39．语句④执行结束后 i=　　。（3 分）

40．语句⑤执行结束后 i=　　。（3 分）

41．语句⑥执行结束后 i=　　。（3 分）

42．语句⑦执行结束后 i=　　。（3 分）

七．设计题（20 分）

43．如下复数类说明了五个成员函数，请详细定义这五个函数的函数体。（每个定义 4 分）

```
class  COMPLEX{                  //real, imag 分别为复数的实部和虚部，
double    real, imag;            //若 x=3+5i，则 x.real=3，x.imag=5
public:
COMPLEX(double r=0, double i=0);
COMPLEX& operator += (const COMPLEX &c);
COMPLEX& operator ++ ( );        //复数的实部和虚部加 1
COMPLEX operator – ( ) const;    //将实部和虚部的符号求反
COMPLEX operator + (const COMPLEX &c) const;
};
```

面向对象程序设计模拟试卷一答案

一解：1. A　　2. D　　3. C　　4. B　　5. C

二解：各类的成员及其存取属性如下。

```
1.private:                    2.private:
        int  a;                       int  d;
  protected:                  protected:
        int  b;                       int  e;
  public:                             int  b;
        int  c;               public:
                                      int  c;
                                      int  f;

3.  private:                  4.private:
        int  g;                     int  n, j;
        int  b;                       int  e, c;
        int  c;                       int  f, h;
  protected:                          int  i;
        int  h;               protected:
  public:                             int  b;
  int  i;                           int  k;
```

三解：main 中每行的输出结果见如下注释。

```
A  a;  cout<<'\n';      //输出 A
B  b;  cout<<'\n';      //输出 B
C  c;  cout<<'\n';      //输出 AC
D  d;  cout<<'\n';      //输出 BACD
E  e;  cout<<'\n';      //输出 AACE
F  f;  cout<<'\n';      //输出 BACACDAACEF
```

四解：错误及其原因见注释。

```
class A{
    static int  a=0;                //不能在类的体内初始化
protected:
    int  b;
public:
    int  c;
    A(int);
    operator int( );
} a(1, 2);                          //没有定义A(int, int)
class B: A{
```

```
        B(int);
        virtual int d;                  //virtual不能用于数据成员
        int    e;
    public:
        A::b;                           //修改而不是恢复访问权限
        friend int operator =(B);       //不存在普通赋值函数的重载，=必须重载为双目的
        static B(int, int);             //构造函数不能定义为不带this参数的静态函数成员
    } b=5;                              //B(int)是不可访问的私有构造函数
    class C: B{
    public:
         int operator++(double);        //必须用int定义后置运算
    };  //C 必须定义构造函数，因 B 定义有带参构造函数且没有无参构造函数
    int main( ){
         int    *A::*p, i;
         i=a.a;          //a.a 是私有的，不能被不是 A 的友元的 main 访问
         i=A(4);
         i=b.c;          //b.c 是私有的，不能被不是 B 的友元的 main 访问
         p=&A::c;        //A::c 不是一个 int *指针
         i=b;            //B 不存在公有的 B::operator int( )或 B::operator A( )
         return ;        //返回 void 类型，与 main 声明的返回类型不一致
    }
```

五解：变量 i 在每条赋值语句执行后的值见注解。

```
         z=b.x;          //i=4
         i=a.x;          //i=4
         i=c.*p;         //i=35
         i=a++;          //i=1
         i=::x+c.y;      //i=37
         i=a+b;          //i=-31
        //A(4,3)++将 A::x=a.x=b.x=c.x 置为 4, 普通成员 a.y=4 不变
         b.h(i)=7;       //i=7
```

六解：本题所涉及的概念并不复杂，值得注意的是分数的约分运算，该运算实际上是一个单目运算。在重载约分运算时，要选择只有一个操作数的运算符。另外，在对分数进行加和乘以后，最好对运算结果进行约分运算，如果不约分不算错。函数成员如下。

```
FRACTION::FRACTION(int num, int den){
    numerator=num;
    denominator=den;
}
int FRACTION::operator>(const FRACTION&f)const{
     return numerator*f.denominator>denominator*f.numerator;
```

```
}
FRACTION FRACTION::operator*( )const{
    int    c=cmd(numerator, denominator);
    return FRACTION(numerator/c, denominator/c);
}
FRACTION FRACTION::operator+(const FRACTION&f)const{
    int    n= numerator*f.denominator+denominator*f.numerator;
    int    d= denominator*f.denominator;
    return *FRACTION(n, d);       //对运算结果进行约分运算
}
FRACTION FRACTION::operator*(const FRACTION&f)const{
    return *FRACTION(numerator*f.numerator,    denominator*f.denominator); //约分
}
```

面向对象程序设计模拟试卷二答案

一解：单项选择题答案

1. B　　2. B　　3. C　　4. C　　5. D

6. D　　7. C　　8. C　　9. D　　10.B

二解：多项选择题答案

11.BCD　　12.ABD　　13.ABC　　14.BC　　15.AB

三解：填空题答案

16. d.a

17. a.b

18. a.c

19. b.d

20. b.b, b.e

21. c.h

22. c.c

23. d.b, d.e, d.h

24. d.B::c, d.f, d.C::c, d.j

25. d.C::c

四解：名词解释答案

26．标识符：标识符是程序变量、常量、函数和类型的名字，通常是由字母或下划线开始的字母、数字或下划线等符号序列构成。（要点：仅回答名字或构成得 2 分，全答得 3 分）

27．重载：重载是指用一个标识符或操作符命名多个函数，用于完成多个不同类型的操作。（要点：没有回答多个扣 1 分，没有回答操作符可不扣分）

28．异常：异常是指函数执行出现了不正常的现象、或运行结果出现了无法定义的情况而必须中断程序执行的状态。（要点：未回答不正（平）常、结果无法定义或中断执行，可扣 1 分）

29．类型强制：类型强制是指将一种类型映射或转换为另一种类型。（要点：没有映射或转换扣 1 分，没有涉及两种类型扣 1 分）

30．抽象类：抽象类是指定义了纯虚函数、或继承了纯虚函数但没有定义函数体的类。或者抽象类是不能定义或产生实例对象的类。（要点：或者前后的两种回答都可以得分）

五解：判断改错题答案

31．✓

32．X　　改正：AC

33．✓

34．✓

35. X　　改正：BACACDAACEF

六解：计算题答案

36. i=4　37. i=4　38. i=35　39. i=7　40. i=37　41. i=47　42. i=7

七解：设计题答案要点如下，达不到要求各扣 1 分。

①所有函数头必须包含 COMPLEX::

②构造函数定义时不能再次定义缺省参数

③返回类型为 COMPLEX&的&不能省略

④返回类型为 COMPLEX&的必须用 return *this

⑤返回类型为 COMPLEX 的必须定义局部对象或使用常量对象

⑥const 不能漏掉，凡尾部部带 const 的函数不能改变当前对象的值。

43. 五个成员函数如下。

```
COMPLEX::COMPLEX(double r, double i){    real=r;    imag=i;    }
COMPLEX& COMPLEX::operator += (const COMPLEX &c){
        real+=c.real;
        imag+=c.imag;
        return *this;
}
COMPLEX& COMPLEX::operator ++ ( ){//前置运算，先运算后取值
        real++;
        imag++;
        return *this;
}
COMPLEX COMPLEX::operator - ( ) const{ return COMPLEX(-real, -imag); }
COMPLEX COMPLEX::operator + (const COMPLEX &c) const
{       return COMPLEX(real+c.real, imag+c.imag);    }
```

面向对象程序设计实践考核

一．考核目的

通过实践考核，检验学生面向对象程序设计基本概念的掌握程度，检验学生对 C++程序设计语言及其面向对象开发环境的熟练程度，考查学生面向对象程序设计与软件开发能力的能力。

二．考核内容

1. 类、封装、继承、多态、抽象类、运算符重载等面向对象的基本概念。
2. 面向对象的分析与设计过程、程序设计的基本方法。
3. 程序设计语言 C++有关类的概念及应用方法。
4. 面向对象的集成开发环境，包括编辑、编译、除错、运行和维护等基本方法。

三．评分标准

满分：100 分，及格：60 分。评分标准如下：

1. 能运用面向对象的开发环境编辑、编译及运行程序，得本题分数的 20%
2. 能运用面向对象的概念编制源程序，得本题分数的 20%
3. 程序的结构和可读性好，得本题分数的 20%
4. 能分析和诊断程序错误，得本题分数的 20%
5. 程序结果正确、注释清晰，得本题分数的 20%

四．样本试题

1. 设计一个表示平面直角坐标系的点的位置 LOCATION 类，提供得到该点坐标、计算两点间距的成员函数。在主程序中创建两个对象 A 和 B，按如下格式输出两个点的坐标和两个点的距离。

A(x1, y1), B(x2, y2), Distance=d

2. 利用 LOCATION 类，不使用继承定义线段类 LINE，提供得到起点和终点坐标的成员函数，以及计算线段长度的成员函数。在主程序中创建一条线段 AB，按如下格式输出起点和终点坐标以及线段的长度。

AB:(x1, y1)--(x2, y2), Length=d

3. 利用 LOCATION 类，使用继承定义圆类 CIRCLE，圆由圆心和半径构成。提供得到圆心坐标和半径的成员函数、以及计算圆的周长和面积的成员函数。在主程序中创建两个圆 A 和 B，按如下格式输出两个圆的圆心坐标、周长和面积，并计算和输出两个圆的圆心之间的距离。

A:(x1, y1, r1), Girth=g1，Area=a1

B:(x2, y2, r2), Girth=g2，Area=a2

A(x1, y1), B(x2, y2), Distance=d

五．考核时间

90 分钟。

面向对象程序设计实践考核答案

1 解:定义 LOCATION 类，该类没有“继承、组成或关联”等现象。注意二维坐标点之间的距离公式。

```
#include <math.h>
#include <iostream.h>
class LOCATION{
    double x, y;
public:
    virtual double getx( )const{ return x; };
    virtual double gety( ) const { return y; };
    virtual double dist(LOCATION &s) const;
    LOCATION(double x, double y):x(x), y(y){ };
};
double LOCATION::dist(LOCATION &s) const {
    double xd=s.x - x, yd=s.y - y;
    return sqrt(xd*xd+yd*yd);
}
int main(int argc, char* argv[])
{
    LOCATION A(0, 0), B(3, 4);
    cout<<"A("<<A.getx( )<<", "<<A.gety( )<<"), ";
    cout<<"B("<<B.getx( )<<", "<<B.gety( )<<"), ";
    cout<<"Distance="<<A.dist(B);
    return 0;
}
```

2 解:线段由起点和终点构成，可面向对象的“组成”关系。通过两个 LOCATION 对象“组成”LINE，注意利用 LOCATION 的距离函数求 LINE 长度。

```
#include <math.h>
#include <iostream.h>
class LOCATION{
    double x, y;
public:
    virtual double getx( ) const { return x; };
    virtual double gety( ) const { return y; };
    virtual double dist(LOCATION &s) const;
    LOCATION(double x, double y):x(x), y(y){ };
```

```
};
double LOCATION::dist(LOCATION &s) const {
    double xd=s.x - x, yd=s.y - y;
    return sqrt(xd*xd+yd*yd);
}
class LINE{
    LOCATION start, end;
public:
    LINE(double sx, double sy, double ex, double ey):start(sx,sy), end(ex,ey){ };
    virtual double getsx( ) const, getsy( ) const, getex( ) const;
    virtual double getey( ) const, length( ) const;
};
double LINE::getsx( ) const { return start.getx( ); }
double LINE::getsy( ) const { return start.gety( ); }
double LINE::getex( ) const { return end.getx( ); }
double LINE::getey( ) const { return end.gety( ); }
double LINE::length( ) const { return start.dist(end); }
int main(int argc, char* argv[ ])
{
    LINE AB(0, 0, 3, 4);
    cout<<"AB("<<AB.getsx( )<<", "<<AB.getsy( )<<")--";
    cout<<"("<<AB.getex( )<<", "<<AB.getey( )<<"), ";
    cout<<"Length="<<AB.length( );
    return 0;
}
```

3 解:圆由圆心和半径构成,可通过“继承“LOCATION 作为圆心,再增加半径定义圆 CIRCLE。注意利用 LOCATION 的距离函数求圆心之间的距离距离。尽管没有使用 class CIRCLE: public LOCATION 形式派生, 但在派生类的函数成员 LOCATION::dist 内部,LOCATION 和 CIRCLE 自动构成父子关系,因此,只要将函数 CIRCLE::dist 的参数定义为父类引用,就可以直接引用函数调用时传递的子类对象。在 CIRCLE::dist 函数中，调用基类的 dist 时一定要用 LOCATION::。否则，根据面向对象的作用域规则，便会调用 CIRCLE 自己的 dist 产生无穷递归。

```
#include <math.h>
#include <iostream.h>
class LOCATION{
    double x, y;
public:
    virtual double getx( ) const { return x; };
    virtual double gety( ) const { return y; };
```

```
    virtual double dist(LOCATION &s) const;
    LOCATION(double x, double y):x(x), y(y){ };
};
double LOCATION::dist(LOCATION &s) const {
    double xd=s.x - x, yd=s.y - y;
    return sqrt(xd*xd+yd*yd);
}
class CIRCLE: LOCATION{
    double    r;
public:
    LOCATION::getx;
    LOCATION::gety;
    virtual double getr( ) const { return r; };
    virtual double area( ) const { return 3.141592654*r*r; };
    virtual double dist(CIRCLE &c) const { return    LOCATION::dist(c); };
    virtual double girth( ) const { return 3.141592654*2*r; };
    CIRCLE(double x, double y, double r): LOCATION(x, y), r(r){ };
};
int main(int argc, char* argv[])
{
    CIRCLE A(0, 3, 2), B(3, 7, 2);
    cout<<"A("<<A.getx( )<<", "<<A.gety( )<<", "<<A.getr( )<<"), ";
    cout<<"Girth="<<A.girth( )<<", Area="<<A.area( )<<"\n";
    cout<<"B("<<B.getx( )<<", "<<B.gety( )<<", "<<B.getr( )<<"), ";
    cout<<"Girth="<<B.girth( )<<", Area="<<B.area( )<<"\n";
    cout<<"A("<<A.getx( )<<", "<<A.gety( )<<"), ";
    cout<<"B("<<B.getx( )<<", "<<B.gety( )<<"), Distance="<<A.dist(B);
    return 0;
}
```