ood

jenny

February 26, 2014

Contents

1 类和对象[1]

1.1 类的定义

C++中的类实际上就是由一组描述对象属性或状态的数据项和作用在这些数据项上的操作构成的封装体。类的定义由关键字class打头，关键 后跟类名，类名之后的括号内是类体，最后以“；”结束。

类与C中的结构大致相似,其不同之处在于:类中规定了哪些成员可以访问,哪些成员不可以访问.这些都通过访问指明赋予以说明.访问指明符有三种:private,protected和public.private使跟着它的成员都私有化,除了该类的成员函数以外,谁也不能访问它们.public则使跟着它的成员公有化,程序中的所有函数(不管是类内定义的还是类外定义的),都可以访问这些成员.下面是stack的

类定义:

```
class stack {
private:
    char v[100];
    char *p;
```

```
public:
    char pop()
    {
        //...
    }
    void push()
    {
        //...
    }
};
```

访问指明符可以作用到跟着它的任意个成员,直到遇到下一个访问指明符.在例中,v和p是私有的,外部函数不能访问它们;pop和push是公有的,任何函数都可以调用它们.由于类中成员缺省的为私有,因此,第一个private可省去.但加上它,能使程序易读.

1.2 对象的定义

与C中的结构类似,定义一个类只是告诉编译器该结构是什么形式,并没有真正的规定存储,也没有创建可用来存放数据的变量.为了预定义存储和创建变量必须提供定义:

```
stack stack1;
```

这个定义创建了stack类的一个实例,类的实例也就是所谓的对象.类的实例具有自己的存储块存放数据和对这些数据实施操作的指令.和用内定义类型定义的变量一样,一个对象在超出定义它的作用域之前始终是存在的(例如,在函数中定义的对象,当函数返回时就撤消了.)并且,类的定义应先于类实例的定义和使用,它们应放在同一源文件中.类的实例也可以用C++的new运算符创建,如以下语句所示:

```
stack *pstack=new stack;
```

这个语句分配一个足以放下该实例的存储块,并返回一个指向该对象的指针,该对象将一直保留分配的存储,直到显式的用delete运算符删除它:

```
delete pstack;
```

一个类可以创建任意多个实例.

在创建类实例之后,就可以访问类中的成员了,类中的成员与结构类似,可以用.和->运算符来访问.但应注意,外部函数不能访问类中的私有成员.以下代码说明对stack类定义的成员、合法与不合法的访问.

```
void main() {
    stack sta; //定义了一个对象stack
    sta.p=sta.v; //错误不能访问私有成员:
    sta.v[1]='c'; //错误不能访问私有成员:
    sta.push('a'); //正确
    char ch=sta.pop(); //正确
}
```

1.3 类的封装性

面向对象程序设计的一个主要特点是封装.C++定义了新的关键字:

pubulic, protected, private.

在定义类时使用这些关键字,可以控制外部能够访问类的成员，访问控制关键字访问权限。

- public: 定义公有成员.所有的对象都可以访问公有成员,类的公有成员构成了类的公共接口.
- protected: 定义保护成员，保护成员只能被类的成员函数,类的友员和公有派生类的成员函数访问，由于和继承有关，这个留到下一章我们再讲.
- private: 定义私有成员.私有成员只能被类的成员函数和类的友员访问.

1.4 构造函数和析构函数

在定义一个类时,直接初始化数据成员是不允许的,因此,下列类定义会产生错误:

```
class c {
private:
    int n=0; //出错
    int rint=n; //出错
    //...
};
```

在类定义中初始化成员本来就没有什么意义,因为类定义只是指出每个成员是什么类型,并不实际预定成员.要想初始化成员,必须要有该类的一个具体实例.为了进行初始化,提供一个公有成员函数是个好办法.如下例所示:

```
void initial() {
    p=v;
}
```

将initial()加入到stack类定义的公有段内,是它能为程序的其它函数调用.现在就可以使用stack类,如下所述:

```
void main() {
    //...
    stack sta;
    sta.initial();
    sta.push('a');
    char ch=sta.pop();
}
```

外面的程序可以利用公有函数inital来初始化类成员,然而,初始化函数仍要用户显式调用.一旦忘记先对类实例进行初始化,有可能带来灾难性的危害.在上例中,如果忘记调用initial操作,那么栈指针p不会指向数组v,它有可能指向无用单元.而接下来的push和pop操作却是通过栈指针p来进行的,这时,程序很可能会异常中止.

初始化数据成员的更好办法是定义一个特殊的成员函数,即所谓构造函数,每当创建一个类的实例时,构造函数的调用自动进行.它可以初始化数据成员,也可以实施一些其他的初始化任务,即为准备该类对象今后使用所要求的任务.

构造函数的名字和类本身的名字相同.在定义构造函数时,不能指定它的返回值,也不必使用void返回值.例如,以下stack类,用构造函数初始化:

```
class stack {
private:
    char v[100];
    char *p;
public:
    stack(){p=v;}
    char pop()
    {
        //...
    }
    push(char ch)
    {
        //...
    }
}
```

注意:如果想创建类的实例,应把构造函数作为公有成员函数,否则,无法创建对象. 有了构造函数,创建stack类的对象时就不必显式初始化了,例如:

```
void main() {
     //...
     stack sta;
     sta.push('a');
     char ch=sta.pop();
 }
```

程序中的

stack sta;

语句要做两件事:它先为sta对象分配空间,然后调用stack的构造函数stack().象这种没有参数的构造函数叫缺省构造函数.缺省的构造函数初始化数据成员时一般是给它们赋缺省值.构造函数也可带有多个参数(任意个).例如,在创建stack类时,希望用户可以指定栈的大小,重写栈类如下:

```
class stack {
    private:
        char *v;
        char *p;
        int size;
    public:
         stack(int sz) {
            p=new char[size=sz]; //动态分配一大小为的字符数组newsz,
            v=p;                 //指向数组中第一个元素的地址p
         }
    //...
};
```

当我们定义它的对象时,采用类似于函数调用的标准语法将参数值传到构造函数:

stack sta(100);

它创建一个stack类的实例,并调用该类的构造函数,把100作为参数传给此函数,因此sta对象中v数组的大小为100.

当类中只有带参构造函数时,如果要创建类对象,必须使用上例所示的带参构造函数,否则编译器会提示出错:

stack sta; //出错

应注意,缺省构造函数和带参数构造函数在定义对象时使用不同语法形式.不要在缺省构造函数之后加上空括号,如果这样作了,实际声明的是一函数,其返回类型是此类,并没定义类的实例:

```
stack sta(); //声明了一个无参数函数
             //返回一个对象stack
```

如果出了这种错,编译器并不会产生错误,直到把sta作为实例用时才出现矛盾. 另外,也可以为带参函数指定缺省值. 当构造函数的所有参数均带缺省值时,既可以把它当作缺省构造函数,也可以把它当作带参构造函数来用.例如,可以把stack公有段内的构造函数改成如下形式:

```
stack(int sz=100)
{
      //...
 }
```

它仍放在stack类的公有段中.定义:

```
stack stack1; //数组大小为100
stack stack2(90);
```

都是合法的. 如某个类定义中没有构造函数,编译器将自动生成一个缺省构造函数,但这种编译器生成的构造函数不会给类的数据成员赋初值.因此,如果想显式初始化数据成员或实施其它某种初始化任务,则必须定义自己的构造函数.一旦定义了自己的构造函数之后,编译器不会再生成一个stack(int sz)构造函数,这时再用 stack stack1; //出错 就会出错.因为编译器找不到缺省的构造函数来初始化它(用户既没定义,编译器也不会自动生成). 与重载全局函数一样,我们也能重载类的构造函数或类的其它成员函数,只有析构函数例外.析构函数是不能重载的,因为它从不带参数.实际上,重载构造函数是极常见的,它们为初始化一个新创建的类对象提供各种变通的方式.例如,下面这个stack的定义就提供了重载构造函数.它既可以指定数据成员的初始值,也可以只用缺省的初始值:

```
class stack {
      private:
         char *v; //栈数组
         char *p; //栈指针
         int size; //栈尺寸
      public:
         //缺省的构造函数
         stack() {
             size=100;
             v=new char[100];
             p=v;
          }
         //带参构造函数
         stack(int sz) {
             v=new char[size=sz];
             p=v;
          }
         //定义其它成员函数
};
```

以下代码演示重载stack()函数的使用:

```
void main() {
      //用缺省构造函数创建一个对象
      stack sta1;
      //创建一个对象并指定栈数组大小
      stack sta2(90);
      //...
 }
```

在使用重载构造函数时应注意不要产生二义性,现在重写stack类中的带参构造函数并为它赋予一缺省值:

```
stack(int sz=70) {
       //...
 }
```

把这个定义放入类stack的public段后,再按下面的形式使用缺省构造函数:

stack sta; //二义性,出错

就会出错.因为在初始化sta时,编译器无法确定是使用缺省构造函数,还是使用具有缺省参数的带参构造函数.

当栈对象是局部变量时(例如,在函数内定义的自动对象),一出作用域,就不能再用它,因此能释放它的所有存储.然而栈中数组v却是用new动态创建的,它只能用delete进行显式删除.在局部变量出作用域时,不会自动释放其存储.因而,在撤消对象时,要调用某函数来释放new所分配的空间．为避免出错,对此种函数的调用应自动进行．为此C++提供了一种特殊的成员函数叫做析构函数,每当要撤消一个类对象时,它就自动调用.析构函数的名字和类的名字也是一样的,只加上一个" ~ "字符的前缀.和构造函数一样,析够函数的定义不得有返回类型(也没有void).但与构造函数不同的是它不接受参数.例如,stack类的析够函数可按如下形式定义:

```
  ~stack()
  {
      delete[] v;
  }
```

它用delete显式删除了用new分配的数组v.

下面我们讨论构造函数和析够函数的调用时间.

一般说来,每当创建一个对象就要调用构造函数;每当撤消一个对象就要调用析够函数.对于某些有特点的对象类型其构造函数和析构函数的调用时间,下面列出准确的说明:

1.对于全局定义的对象(就是说定义在所有函数之外),每当程序第一次运行,主函数main接受控制之前,就要调用构造函数.整个程序结束时调用析构函数.

2.对于局部定义的对象(也就是在一个函数内),每当程序控制流到达该对象定义处则调用构造函数;每当程序控制走出定义该对象的程序块则调用析构函数(也就是对象出了作用域).

3.对于用关键字static定义的局部对象,当程序控制第一次到达该对象定义处则用构造函数,整个程序结束时调用析构函数.

4.对于用new运算符动态创建的对象,每当创建该对象时,调用构造函数;每当用delete运算符显式的撤消对象时,调用析构函数(如程序员不显式撤消该对象,是不会调用析构函数的).

1.5 初始化表

一个数据成员可以被定义为另一个类的对象,即一个类的对象中嵌套另一个类的对象.这种数据成员就是所谓的成员对象.成员对象是类实例的一部分.在创建类对象时,也要创建它.同样,当用构造函数初始化嵌套对象时,也要用成员对象的构造函数来初始化成员对象.如果成员对象的构造函数带有参数,则需通过成员初始化符表把参数传到成员对象的构造函数中．成员初始化符表的位置紧跟着构造函数定义的参数表之后,由冒号,接着是一个或多个用逗号隔开的成员初始化符组成.成员初始化符由数据成员的名字和括在括号内的初始值组成.初始化成员对象是将它所要求的参数传递到该对象的构造函数,而它的构造函数在包容类的构造函数的成员初始化符表之中.例如,以下代码中的类CContainer包容了一个类的CEmbedded的成员对象,该对象在对象在CContainer的构造函数中初始化:

```
 class CEmbedded {
       //...
       public:
          CEmbedded(int Parml,int Parm2) {
                    //...
           }
```

```
        //...
};

class CContainer {
    private:
        CEmbedded embedded;
    public:
        CContainer(int p1,int p2,int p3):embedded(p1,p2) {
                //...
         }
    //...
 };
```

如果在构造函数的成员初始化符表中没有初始化成员函数(或者那个构造函数本身就是由编译器生成的缺省构造函数),则编译器就会自动引入成员函数的缺省构造函数,只要它可用(再重申一次,并非每个类都有缺省的构造函数,只要缺省构造函数不可用就会出错).

利用成员初始化符表可以初始化其它数据类型.特别是常量和引用,在构造函数中不能用赋值方式来初始化它们,例如:

```
class C
{
    private:
        int n;
        const int cInt;
        int &rInt;
        //...
    public:
        C(int pram):n(parm),cInt(5),rInt(n)
        {
            //...
         }
    //...
 };
```

下面的定义创建了一个对象,其数据成员n和cInt被初始化为0和5,数据成员rInt被初始化为n的别名: C cobject(0);

2 类和对象$_2$

2.1 This指针

当引用一个类的数据成员其引用代码又在类外时,在表达式中总要指明该类一个具体实例,编译器才能知道要访问哪一个数据成员.例如,以下代码首先打印属于对象test1的数据成员n,然后打印属于对象*ptest2的n,CTest是一个类:

```
CTest test1;
CTest *ptest2=new CTest;
//...
cout<<test1.n<<'\n';
cout<<ptest2->n<<'\n';
```

然而,在一个类的成员函数内部引用一个数据成员却不能指明一个类的具体实例:

```
class CTest
{
    public:
        int n;
        int getn()
        {
              return n;
         }
 };
```

那么,编译器怎样决定引用哪一个实例对象的n拷贝呢?为了分辨,编译器实际上给成员函数传递一个隐藏的对象指针.此指针指向函数调用所要引用的对象.该函数隐式的使用这个指针,例如,在以下调用:

```
CTest test;
test.getn();
```

中,编译器传给getn一个隐藏的指向test对象的指针,getn隐式的使用这个指针,访问属于对象test的成员n.

利用C++的关键字this可以直接访问这个隐藏的指针.事实上,对于类X,它的每一成员函数中都隐式的声明了this指针:

```
X * const this;
```

此指针指向该成员函数所在的对象的地址(即函数"当前"引用的对象).在test.getn()调用中,getn()中的this指针指向的就是test对象.而对于

```
CTest test1;
test1.getn();
```

getn()中的this指针,指向的是test1对象.

由于this指针被声明为* const,它是一个指针常量,因此不能改变它的值(在低版本的C++中,修改this指针是可以的),但可以改变它所指对象的值.例如:

```
int CTest::chargethis() //假定在中已有声明changethisCTest
{
      CTest temp;
      this=temp; //非法
      //...
 }
```

利用this指针,getn可重写如下:

```
int CTest::getn()
{
      return this->n; //等价于return n;
}
```

在数据成员的名字前加上表达式this->是合法的但没有什么效果,因为this指针的使用本来就是隐式的,它只引用数据成员.

如果需要访问全局数据或函数,而它们的名字又和数据成员或成员函数的名字相同,则必须在名字前面加上作用域分辨符" :: ".例如:

```
int n=0; //全局n
class CTest
{
      int n; //数据成员n
      int demo()
      {
            cout<<::n<<'\n'; //打印全局n
            cout<<n<<'\n'; //用访问数据成员 thisn
      }
};
```

2.2 new和delete运算符

new和delete是C++新引入的单目运算符,它们可以从堆上分配和删除存储块(堆在C++中也叫自由存储).用new运算符时要指明数据类型,以后new就分配一个足以放下指明类型对象的存储,并返回该存储块的首地址作为指向指定类型的指针.在下面,我们用new运算符为内定义类型分配存储:

```
char *pChar; //声明一个指针
int *pInt;
double *pDouble;
pChar=new char; //分配存储对象
pInt=new int;
pDouble=new double;

*pChar='a'; //赋值
*pInt=5;
*pDouble=2.5;
```

若new运算符不能分配所要求的存储,它返回零值,因此,在使用返回的指针时应检查它:

```
pInt=new int;
if(pInt==0)
      //处理错误条件
else
     //可以用了pInt
```

当以new运算符分配的存储块用过之后,我们可以用delete运算符作用于指向该地址的指针,删除该存储.例如,以下语句将删除在上例中分配的存储块.

```
delete pChar;
delete pInt;
delete pDouble;
```

在前面,我们曾经用变量定义的方式来生成对象.这种对象的个数和大小在编译时就确定了,在运行之前无法得知对象的个数和大小时,采用变量定义形式并不方便,这时可以利用new运算符动态的生成对象.动态对象的个数和大小都在运行时予以确定,动态对象的撤消也可以通过显式的调用delete加以控制.

一般说来,new运算符和delete运算符比C中传统的存储分配函数malloc和free更有用. new运算符可以根据对象的类型,自动的决定某个对象的大小,而malloc则要显式指定要分配的存储空间大小.new运算符返回的是一个指向正确类型的指针,malloc返回的是一个void *类型的指针.因此在使用new时不必进行强调类型转换,它是类型安全的,而malloc需要强制类型转换,可能会带来错误.

我们也可以利用new运算符来为数组分配空间,在分配时,先得指定一个基本类型(即数组元素的类型),再在[]字符内指定它的元素个数,如下例所示:

```
int size;
cin>>size; //输入数组的大小
char *string=new char[25]; //创建有个字符的数组25
int * arrayInt=new int[size]; //创建大小为的数组sizeint
double *arrayDouble=new double[32]; //创建有个32型元素的数组double
//...
```

在分配一个数组时,new返回数组第一元素的地址.注意,由于new进行动态分配,可以用变量,如size指定数组元素的个数, 这个变量的值具体是多少在运行时才能确定.

在用delete删除一个数组时,必须包含一对空[]字符来指示要撤消的是一个数组,而不是该基类型的简单对象.例如,可以用以下语句来撤消上例中分配的数组:

```
delete []string;
delete []arrayInt;
delete []arrayDouble;
```

用new分配的存储块不能自动初始化为0,但用new分配内定义类型对象的存储(如一个char)时,可以用相应的值显式初始化该对象,其语法如下:

```
char *pChar=new char('a'); //用'a初始化'char
int *pInt=new int(3); //用初始化3int
```

上面的两个语句不仅为char和int分配了空间,还把它们分别初始化为'a'和3.

2.3 静态类成员

在常情况下,一个类的各个实例都有它们自己的、属于该类的数据成员和私有拷贝.然而, 若用关键字static声明数据成员,则该数据项的简单拷贝,无论该类创建了多少实例,它始终是存在的(即使类的实例一个也没有创建).例如,以下类定义就定义了静态数据项count:

```
class CTest
{
    public:
    static int count;
    //...
 };
```

无论CTest类创建了多少实例,count将严格只存放一个拷贝.

除此而外,如果在类中声明了静态数据成员,必须象全局数据项一样,在类外定义和初始化它.由于静态数据成员的定义在类外出现,则必须用作用域分辨符在声明中说明它是哪个类的(本例是CTest::),定义和初始化count可如下例示:

```
int CTest::count=0;
```

由于静态数据成员独立于任何类对象存在,在定义时,用类名和作用域分辨符就可以访问它,无需引用类实例.可以把静态数据成员设想为全局变量和该类的正常数据成员的综合.与全局变量一样,它在函数之外定义和初始化,它表示的单个存储单元和整个程序作业一样持久.与正常的数据成员一样,它在类中声明且作用域仅限于这个类,其访问也是受控的(即,它可以是公有、私有、受保护的成员).

成员函数也可以用static关键字修饰,如下例所示:

```
class CTest
{
      //...
      static int getCount()
      {
           //...
       }
};
```

2.3.1 静态成员函数有以下性质:

1. 类外的代码可利用类名和作用域分辨符调用这个函数,无需引用一个类的实例(甚至类实例可以不存在),如下例示:

```
1   void main()
2   {
3          int count=CTest::getCount();
4          //...
5    }
```

2. 静态成员函数只可以引用属于该类的静态数据成员或静态成员函数(因为它无需引用一个类实例就可以被调用,静态成员函数没有this指针存放对象的地址.因此,如果它试图访问一个非静态数据成员,编译器无法判定它所访问的数据成员是哪个对象的).

   静态数据成员和静态成员函数可用于保持用于类的数据项或一个类的所有实例共享的数据项.以下程序说明了以静态成员保持一个类的实例当前个数的计量:

```
1  #include<iostream.h>
2
3  class CTest {
4  private:
5      static int count;
6  public:
7      CTest() {
8          ++count;
9      }
10
11     ~CTest() {
12         --count;
13     }
14
15     static int getCount() {
16         return count;
17     }
18 };
19
```

```
int CTest::count=0;

void main() {
    cout<<CTest::getCount()<<"object exist\n";
    CTest test1;
    CTest *ptest2=new CTest;
    cout<<CTest::getCount()<<"object exist\n";
    delete ptest2;
    cout<<CTest::getCount()<<"object exist\n";
}
```

本程序打印如下: 1 object exist

2.4 友员

类可以给予另一函数或类存取其私有成员的权力.这样的存取必须将别的类或非成员函数声明为友员.友员当作类成员对待,并且对对象私有区的存取没有限制.如果要用友员来访问类,友员应在此类中声明.声明的形式是在普通声明之前加上friend关键字.应记住,友员无论是在公有区还是在私有区声明都没关系,它只是将友员的名字引入了类作用域.例如:

```
class t1
{
    private:
          int data;
          friend void friend_t1(t1 fri); //友员声明
    public:
          t1(){data=12;}
           //...
};
```

t1声明了外部函数$\text{friend}_{t1}()$是类t1的友员,$\text{friend}_{t1}()$的定义如下:

```
void friend_t1(t1 fri)
{
   fri.data=10; //改变了的私有变量t1t1::的值data
}
```

由于它是类t1的友员,因此可修改类t1中的私有数据.

应注意,友员并不是成员,因此在成员函数中,不能用this指针来访问它.现在假定在t1中加入一成员函数:

```
  void t1::use_friend()
  {
          t1 fri;
          this->friend_t1(fri); //出错,不是类成员friend_t1
          ::friend_t1(fri); //正确访问外部函数,friend_t1
   }
```

类似的,在外部函数中,也不能通过t1的对象来访问友员$\text{friend}_{t1}()$:

```
   void main()
   {
      ti fri,fri1;
      fri.friend_t1(fri); //出错,不是的成员函数friend_t1t1
      friend_t1(fri1); //正确调用外部函数,friend_t1
   }
```

一个类的成员也可以声明为另一个类的友员,例如:

```
class x{
      public:
          void f();
          //...
      private:
          int i;
          //...
};

class y{
    friend void x::f();
    int i;
    public:
    //...
};
```

y声明x的成员函数f()是它的友员,故x::f()可访问y中的私有变量:

```
void x::f()
{
      y yi;
```

```
        yi.i=10; //修改了类的对象的私有成员yyi
    }
```

与外部友员函数类似,类成员友员只是它所在类的成员函数,而不是声明它为友员的那个类的成员函数.例如:

```
y y1;
y1.x::f(); //x::f不是()的成员函数y出错,
x x1;
x1.f(); //正确,x::f是()的成员函数x
```

甚至可以将整个类声明为另一个类的友员:

```
class y
{
    //...
};
class x
{
    friend class y;
    //...
};
```

\end{listings}由于有了这个友员声明类
,的任何成员函数都可以访问类的私有成员yx.友员破坏了封装原本只能通过公有接口间接访问的私有数据现在可以由友员直接访问因此友员要了解它所访问的类的实现细节，当此类的实现有所改动，即使公共接口不改变，友员也要跟着改动。声明有友员的类是难修改的例如

```
,,.,,:
\begin{lstlisting}[language=c++]
class sf; //由于在定义之前sf,要用到它stack先声明,sf
class stack
{
    char *p; //栈指针
    char *v; //栈数组
    int sz;
public:
    stack(int size);
    char pop();
    void push(char);
    ~stack();
    friend void sf::f(); //sf::是的友员fstack
};

class sf
{
    //...
    void f()
    {
        //使用stack::和pstack::v
    }
};
```

stack声明了sf::f()是它的友员.现在,如果要把stack的数据结构改为链表形式,它不仅要修改pop()和push()的实现,还要修改类sf中f()的实现.也就是说修改从一个模块波及到了另一个模块,而不只限于模块内部.

友员能破坏封装,但也能使程序变得简洁而高效.在具体使用时,应权衡二者,再做适当选择.

3 类和对象2,

类(class)

是C++面向对象程序设计的核心，它是实现抽象类型的工具。类是通过抽象数据类型的方法来实现的一种数据类型。类是对某一类对象的抽象，而对象是某一种类的实例。

本文稍微说一下类和对象的基本使用，比较简单也是最基本的C++基础。

3.1 类

类的定义有2个部分：声明部分和实现部分。

声明部分：用来声明类中的数据成员和成员函数(方法)。->告诉用户"干什么？"。

实现部分：用来对成员函数的定义。->告诉用户"怎么干"。

3.2 类的关键字

1.public: 公有成员，在程序的任何地方都可以被访问。

2.private: 私有成员，只能被成员函数和类的友元访问。

3.protected: 保护成员，用于继承中。

3.3 类的声明

下面只提供最简单的类声明，因为一个真正的类，还会涉及到很多性质或者特点。

本实验是在Code::Blocks工具中进行的，为了规范性，这里总共创建了3个源文件：main.cpp test1.cpp test1.h

main.cpp: 主程序

test1.cpp: test1.h头文件中的类的定义

test1.h: 类的声明

```
//test1.h
#ifndef TEST1_H_INCLUDED
#define TEST1_H_INCLUDED

class CAnimal
{
public:
    int getAge();
    void setAge(int age);
    void addAge();

private:
    int mAge;
};

#endif // TEST1_H_INCLUDED
```

3.4 类的定义

当类的成员函数的函数体在类的外部定义时，必须由作用域运算符"::"来通知编译系统该函数所属的类。

```
//test1.cpp
#include "test1.h"

int CAnimal::getAge()
{
    return mAge;
}

void CAnimal::addAge()
{
    mAge++;
}

void CAnimal::setAge(int age)
{
    mAge = age;
}
```

3.5 对象的定义

CAnimal类的概念已经创建出来，现在就要实例化出一只动物，这里就用阿猫阿狗来举例。以下用2种方式来实例化对象:

1.CAnimal dog; //创建出了一只dog

2.CAnimal *cat = new CAnimal; //动态创建了一只cat

以下贴出使用dog和cat的具体操作:

```
//main.cpp
#include <iostream>
#include "test1.h"

using namespace std;

int main()
{
    //dog
    CAnimal dog;

    dog.setAge(12);
    cout << "Dog␣is␣" << dog.getAge() << "␣years␣old." << endl;

    cout << "After␣one␣year." << endl;

    dog.addAge();
    cout << "Dog␣is␣" << dog.getAge() << "␣years␣old." << endl;
```

```
    //cat
    CAnimal *cat = new CAnimal;
    cat->setAge(12);
    cout << "Cat is " << cat->getAge() << " years old." << endl;

    cout << "After one year." << endl;

    cat->addAge();
    cout << "Cat is " << cat->getAge() << " years old." << endl;

    return 0;
}
```

3.6 程序运行结果

```
Dog is 12 years old.
After one year.
Dog is 13 years old.
Cat is 12 years old.
After one year.
Cat is 13 years old.
```

3.7 C++中class与struct的区别

在C++面向对象语言中，class和struct的功能很相似，struct也是一种类。

区别:

一个struct也是一个class，但是struct的默认方式是公用部分，而class的默认方式是私用部分。

以下使用代码说明，下面的class与struct是等价的:

```
1 :
class
class CAnimal
{
    //默认是private
    int mAge;
public:
    int getAge();
    void setAge(int age);
    void addAge();
};
<=>:
struct
struct CAnimal
{
    //默认是public
    int getAge();
    void setAge(int age);
    void addAge();
private:
    int mAge;
};
```

3.8 为什么类的定义以分号结束

这是什么问题? 有人说我很无聊。

其实并不单单类的定义要以分号结束，C++中的struct也要以分号结束。原因很简单，因为咱们的使用习惯，或者说是编译器编译程序要考虑的情况，在定义之后，可以接一个对象的定义列表，理所当然要以分号结束。

看几个简单的例子:

```
class CAnimal
{
//...
}dog, cat;

struct CAnimal
{
//...
}dog, cat;

int a, b, c; //0∩(∩_)哈哈0~
```

4 构造函数

C++规定：构造函数必须与相应的类同名，它可以带参数，也可以不带参数，与一般的成员函数定义相同，而且可以重载，构造函数的重载跟普通的成员函数的重载是一样的。

4.1 构造函数的重载

上面的例子使用了：void printInfo() const;

const在这里的作用是，保证在printInfo函数体中，不更改变量的值，若不小心设置了某变量的值，编译器会在编译的第一时间提醒用户程序出错，提高了开发效率。

4.2 带默认参数的构造函数

构造函数与成员函数，都可以设置形参的默认参数。如果构造函数或成员函数在类中的声明初始化了默认参数，那么在类外定义的时候，形参不能写出默认值，具体看以下例子。

```
/*带默认参数的构造函数*/
#include <iostream>

using namespace std;

class CDate
{
public:
    CDate(int year, int month = 2, int day = 8);
    ~CDate();
    void printInfo() const;
private:
    int mYear;
    int mMonth;
    int mDay;
};

CDate::CDate(int year, int month, int day)
{
    cout << "调用构造函数" << endl;
    mYear = year;
    mMonth = month;
    mDay = day;
}

CDate::~CDate()
{
    cout << "调用默认析构函数" << endl;
}

void CDate::printInfo() const
{
    cout << "Date:␣"
         << mYear << "年"
         << mMonth << "月"
         << mDay << "日"
         << endl << endl;
}

int main()
{
    CDate day1(2012);
    day1.printInfo();

    CDate day2(2012, 3, 15);
    day2.printInfo();

    return 0;
}
```

执行结果：

```
[plain] view plaincopy调用构造函数

Date: 年月日201228调用构造函数


Date: 年月日2012315调用默认析构函数调用默认析构函数
```

4.2.1 上面的例子是用了：CDate(int year, int month = 2, int day = 8);

被设置默认参数的形参都是靠后面的参数，那如果是：CDate(int year=2012, int month = 2, int day);这样可以吗?

经过编译器验证，这样声明是错误的。在声明一个对象时，一般都会省略后面的参数，而不会省略靠前面的参数，不然就乱套了。所以从这里可以得出一个概念：在构造函数或成员函数中，被设置默认参数的形参，后面绝对不允许出现没有设置默认参数的形参。

以CDate(int year, int month = 2, int day = 8)为例，设置默认参数有以下几种情况：

1. 全部给形参设置默认参数 CDate(int year=2012, int month = 2, int day = 8); 正确
2. 给一部分形参设置默认参数
   CDate(int year, int month = 2, int day = 8); 正确
   CDate(int year, int month, int day = 8); 正确
   有且只有这3种情况出现，比如以下声明均是错误的：
   CDate(int year, int month = 2, int day); 错误
   CDate(int year=2012, int month = 2, int day); 错误

4.2.2 带默认参数的函数的定义

1. 构造函数的声明与定义分开，在类中声明：
   CDate(int year, int month = 2, int day = 8);
   在类外定义：

```
CDate::CDate(int year, int month, int day)
{
    cout << "调用构造函数" << endl;
    mYear = year;
    mMonth = month;
    mDay = day;
}
```

   在函数定义的时候，那些默认参数就不必写出来了。

2. 在类中声明与定义
   如果直接在类中声明并定义带默认参数的构造函数，直接定义：

```
CDate(int year, int month = 2, int day = 8)
{
    cout << "调用构造函数" << endl;
    mYear = year;
    mMonth = month;
    mDay = day;
}
```

5 类对象的复制控制

前面学习了构造函数的用法与初始化列表的特性，本章还是学习构造函数的内容。类型还能控制复制、赋值或撤销该类型的对象，这时候，类通过特殊的成员函数：复制构造函数、赋值操作符和析构函数来控制这些行为。这些都属于类对象的复制控制，挺重要的一部分内容。

5.1 本文内容的应用

5.1.1 复制构造函数的例子：

```
string str1("hello");
    string str2(str1);
```

5.1.2 赋值操作符的例子：

```
string str1("hello");
    string str2 = str1;
```

5.2 复制构造函数的应用于哪些地方

什么样的函数才是复制构造函数?

只有单个形参，而且该形参是对本类类型对象的引用(常用const修饰)。

复制构造函数用于：

5.2.1 用一个同类型的对象显示或隐式初始化一个对象。

eg:

```
    string str1("hello");
    string str2(str1);
```

5.2.2 作为函数的实参

eg:

```
void Show(string str);
...
string str1("hello");
Show(str1); //存在一个复制操作，将str1对象隐式复制到Show函数的临时对象
```

5.2.3 作为函数的返回值

eg:

```
string getString() const
{
string str("hello");
...
return str;
}
...
string str1 = getString(); //将函数的返回值的对象复制到对象getStringstringstr1
```

5.2.4 初始化顺序容器的元素

eg:

```
vector<string> vec(5); //声明了一个，包含个对象；vector5string
//编译器首先使用默认构造函数创建一个临时对象来初始化，然后使用复制构造函数将临时值复制到的每个元素中。stringvecvec
```

5.2.5 根据元素初始化式列表初始化数组元素

这里要强调的是：初始化式列表

1. 每个元素调用一次默认构造函数
   string str[1]; //调用了10次默认构造函数
2. 每个元素调用一次复制构造函数

```
1 string str1, str2, str3, str4;
2 string str[] = { str1, str2, str3, str4};
```

5.3 合成的复制构造函数和合成的赋值操作符

如果用户没有显示的定义复制构造函数或赋值操作符，编译器将会默认的合成一个复制构造函数。复制构造函数与合成复制构造函数，或者赋值操作符与合成赋值操作符不同，编译器默认合成的函数的行为是"逐个成员初始化"，将新对象初始化为原对象的副本。逐个成员初始化不包括static成员，只负责初始化非static成员。

5.4 小试牛刀

这里编写一个最简单的例子，来展示一下类对象的复制控制。

```
#include <iostream>
using namespace std;

class CObj   {
public:
    CObj() {
        cout << "默认构造函数" << endl;
    }
    CObj(const CObj &obj) {
        cout << "复制构造函数" << endl;
    }
    CObj & operator=(const CObj &obj) {
        if (&obj != this) {
            cout << "赋值操作符" << endl;
        }
        return *this;
    }
};

int main()   {
    CObj obj1;
    cout << endl;
```

[1]DEFINITION NOT FOUND.

```
    CObj obj2(obj1);
    cout << endl;

    CObj obj3;
    obj3 = obj2;
    return 0;
}
```

执行结果:
默认构造函数
复制构造函数
默认构造函数
赋值操作符

5.5 禁止复制

5.5.1 有些类需要完全禁止复制，怎么办?

只需将复制构造函数声明为私有成员函数即可。
解释：因为如果不声明定义，编译器将会默认合成一个复制构造函数。
例子:

```
#include <iostream>
using namespace std;

class CObj   {
public:
    CObj() {
        cout << "默认构造函数" << endl;
    }
private:
    CObj(const CObj &obj) {
        cout << "复制构造函数" << endl;
    }
};

int main()   {
    CObj obj1;
    CObj obj2(obj1);

    return 0;
}
```

编译报错:
error: 'CObj::CObj(const CObj&)' is private
这就是禁止复制的效果，正是这里想要的。

5.5.2 连友元函数和成员函数的复制也要禁止，怎么办?

如果想要连友元函数和成员函数的复制也禁止，那么可以声明一个私有的复制构造函数但不对其定义。
解释：因为友元函数或成员函数可以访问到类的私有成员，所以当然能调用私有的复制构造函数，所有将复制构造函数声明为私有但不定义，就能避免友元或成员函数的调用。

```
#include <iostream>
using namespace std;

class CObj   {
public:
    CObj() {
        cout << "默认构造函数" << endl;
    }

friend void fcopy();
private:
    CObj(const CObj &obj);
};

void fcopy()   {
    CObj obj1;
    CObj obj2(obj1);
}

int main()   {
    fcopy();
    return 0;
```

```
}
```

编译报错:
undefined reference to 'CObj::CObj(CObj const&)'
这也正是这里需要的结果，禁止复制成功，OK。

6 静态成员

静态成员的提出是为了解决数据共享的问题。

6.1 静态数据成员

静态数据成员是同一个类中所有对象共享的成员，而不是某一对象的成员。因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。

6.1.1 如何定义一个静态数据成员

1. 使用关键词static声明静态数据成员。(在类中)
2. 对静态数据成员进行初始化。(在类外) 由于静态数据成员要分配空间，因此不能在类声明中进行初始化。静态数据成员初始化在类的外部进行，与一般数据成员的初始化不同。
   static数据成员的初始化格式:
   <数据类型> <类名>::<静态数据成员> = <值>;
   静态数据成员的使用例子:

```
#include <iostream>
using namespace std;

class CObj {
public:
    CObj() {
        mObjs++;
    }
    static int GetObjs() {
        return mObjs;
    }

private:
    static int mObjs; //声明静态数据成员
};

int CObj::mObjs = 0; //静态数据成员的初始化

int main() {
    CObj objs[10];
    cout << "类总共创建了CObj" << CObj::GetObjs() << "对象" << endl;
    return 0;
}
```

执行结果:
CObj类总共创建了10对象

6.2 静态成员函数–>(没有this指针)

静态成员函数和静态数据成员一样，它们都属于类的静态成员，都不是对象成员。因此对静态成员的引用不需要用对象名，不过用对象名来引用也是可以的。

注：在静态成员函数的实现中，不能直接引用类中的非静态成员，可以引用类中说明的静态成员。但静态成员函数中可以间接的引用非静态成员，可以通过对象来引用。

来个例子，说明在 静态成员函数中，可以直接引用静态数据成员，但不能直接引用非静态数据成员，非静态数据成员必须通过对象来引用。

```
#include <iostream>
using namespace std;

class CObj {
public:
    CObj() {
        mObjs_static++;
    }
    static void ShowData(CObj obj) {
        //mObjs_nonstatic = mObjs_static; 错误，不能直接引用非静态成员//
        obj.mObjs_nonstatic = mObjs_static;
        cout << "直接引用:␣" << "类总共创建了CObj" << mObjs_static << "对象" << endl;
        cout << "间接引用:␣" << "类总共创建了CObj" << obj.mObjs_nonstatic << "对象" << endl;
    }
```

```
private:
    static int mObjs_static; //声明静态数据成员
    int mObjs_nonstatic;
};

int CObj::mObjs_static = 0; //静态数据成员的初始化

int main() {
    CObj obj1, obj2, obj3;
    obj1.ShowData(obj1);
    cout << endl;
    CObj::ShowData(obj2);
    return 0;
}
```

执行结果:
直接引用: CObj类总共创建了3对象
间接引用: CObj类总共创建了3对象
直接引用: CObj类总共创建了3对象
间接引用: CObj类总共创建了3对象

6.3 const的static成员

上面已经说了，类的static数据成员，不能在类的声明中初始化，只能在类外进行初始化。但这里存在一种特殊情况，如果该静态数据成员是一个常量表达式，那么它就能够在类中进行初始化。

初始化的格式是：static const <数据类型> <变量名> = <值>;

```
#include <iostream>
using namespace std;

class CObj
{
public:
    static const int State_Mount  = 0;
    static const int State_Umount = 1;
};

int main()
{
    cout << "State_Mount␣␣=␣" << CObj::State_Mount << endl
         << "State_Umount␣=␣" << CObj::State_Umount << endl << endl;

    CObj obj;
    cout << "State_Mount␣␣=␣" << obj.State_Mount << endl
         << "State_Umount␣=␣" << obj.State_Umount << endl;
    return 0;
}
```

执行结果:
$\text{State}_{\text{Mount}} = 0$
$\text{State}_{\text{Umount}} = 1$
$\text{State}_{\text{Mount}} = 0$
$\text{State}_{\text{Umount}} = 1$

7 继承与派生类

#+继承的规则

继承方式	基类特性	派生类特性
公有继承(public)	public	public
	protected	protected
	private	不可访问
私有继承(private)	public	private
	protected	private
	private	不可访问
保护继承(protected)	public	protected
	protected	protected
	private	不可访问

8 多态性与虚函数

C++的面向对象思想博大精深，现在到了面向对象的多态性。
强烈推荐：C++编程思想(第2版) 第1卷 标准C++引导 第15章 多态性和虚函数
我觉得这章讲得很清楚，包括编译器对虚函数的实现，还有C++的多态性，都挺不错的。看书就得取我所需，以免浪费太多时间。

8.1 多态性

多态性是指不同类型的对象接受相同的消息时产生不同的行为。在C++中，多态性可以分为两种：编译时的多态性和运行时的多态性。
编译时的多态性：函数的重载或运算符的重载；
运行时的多态性：虚函数来实现。
函数的重载和运算符的重载将在后面的博文中讨论，本文主要是讨论虚函数的使用。

8.2 动态绑定(晚捆绑)

动态绑定这词比较专业，在《C++ Primer中文版》书中看到。简单的说，动态绑定就是在程序运行过程中才绑定到具体的函数，编译器在编译过程中还无法确定要调用的具体类型对象的函数，所以只能动态绑定。

8.2.1 在C++中，要触发动态绑定，必须满足2个条件：

1. 只有指定为虚函数的成员函数才能进行动态绑定，使用virtual关键字修饰；
2. 必须通过基类类型的引用或指针进行函数调用。
   这2点必须记住，非常重要的两点，下文会使用一个例子来说明，但在看例子之前，先讨论一下基类与派生类之间的转换。

8.3 派生类到基类的转换(将派生类转换为基类)

将派生类转换为基类是比较好理解的，因为每个派生类对象都包含基类部分，所以可将基类类型的引用绑定到派生类对象的基类部分，或者用指向基类的指针指向派生类对象。

8.3.1 引用转换不同于转换对象

1. 引用转换：引用直接绑定到该对象；
2. 对象转换：使用派生类对象的基类部分对基类对象进行初始化和赋值，派生类部分被切掉，简称"对象切片"。

8.3.2 派生类到基类转换的可访问性

派生类到基类的转换可能不可访问，具体规则如下(当然这是不太准确的，只是稍微提一下，具体在《C++ Primer中文版》书中)：
public继承：可以转换
private或protected继承：不可转换

8.4 基类与派生类的转换(将基类转换为派生类)

就一句话：从基类到派生类的自动转换时不存在的，也是不安全的。
原因：基类对象只能是基类对象，它不能包含派生类型的成员。

8.5 再谈动态绑定

结合以上几点，写个小例子就能很清楚的理解多态性的概念和使用。一般我写例子都是小儿科级别的，只是体现一下概念和使用方法。

```
#include <iostream>
using namespace std;

class CBase
{
public:
    virtual void func() const
    {
        cout << "CBase::func" << endl;
    }
};

class CDerived : public CBase
{
public:
    void func() const
    {
        cout << "CDerived::func" << endl;
    }
};

void test(CBase &obj)
{
    obj.func();
```

```
}

int main()
{
    CBase base;
    CDerived derived;
    test(base);
    test(derived);
    return 0;
}
```

执行结果：
CBase::func
CDerived::func
程序要求两点：

8.5.1 只有指定为虚函数的成员函数才能进行动态绑定，使用virtual关键字修饰；

论证：
如果去掉基类func函数的virtual声明，那么执行结果：

```
CBase::func
CBase::func
```

8.5.2 必须通过基类类型的引用或指针进行函数调用。

论证：
如果将"void test(CBase &obj)"改成"void test(CBase obj)"，不使用引用，那么执行结果：

```
CBase::func
CBase::func
```

所以上述两个条件必须满足，才能够使用虚函数这一特性。使用虚函数，效率肯定会比较低，因为调用函数得需要2步，根据具体的对象的引用或指针找到实际的对象类型，然后再计算虚函数表中要调用的函数地址。但C++的成员函数默认不是虚函数，只能通过这两个条件来触发，所以给程序员提供了灵活性。
如果将基类中的成员函数声明为虚函数，那么使用基类的引用或指针来传递基类或派生类的对象时，将会根据引用或指针实际指向的对象类型来调用相应类中的函数。
再嗦：C++编程思想(第2版) 第1卷 标准C++引导 第15章 多态性和虚函数 这章介绍得很清楚。

8.6 虚析构函数

如果用基类类型的引用或指针绑定到派生类的对象，那么基类的析构函数将只能调用到基类自身的析构函数。但如果把基类中的析构函数也声明为虚函数，那么情况就完全不同了，基类的虚析构函数根据基类类型的引用或指针实际绑定的对象类型来调用析构函数。在基类的析构函数声明为虚函数的情况下，若基类的引用绑定到派生类的对象，则在销毁对象的时刻，基类将会动态绑定到派生类对象，调用派生类的析构函数进行清除工作，随后再调用自身的析构函数。
来个小例子，就能看出区别了：

```
#include <iostream>
using namespace std;

class CBase1
{
public:
    ~CBase1()
    {
        cout << "CBase1::~Cbase1()" << endl;
    }
};

class CDerived1 : public CBase1
{
public:
    ~CDerived1()
    {
        cout << "CDerived1::~CDerived1()" << endl;
    }
};

class CBase2
{
public:
    virtual ~CBase2()
    {
        cout << "CBase2::~Cbase2()" << endl;
```

```
    }
};

class CDerived2 : public CBase2
{
public:
    ~CDerived2()
    {
        cout << "CDerived2::~CDerived2()" << endl;
    }
};

int main()
{
    CBase1 *base1 = new CDerived1;
    cout << "析构函数" << endl;
    delete base1;
    CBase2 *base2 = new CDerived2;
    cout << "虚析构函数" << endl;
    delete base2;
    return 0;
}
```

执行结果:

```
1  析构函数

CBase1::~Cbase1()虚析构函数

CDerived2::~CDerived2()
CBase2::~Cbase2()
```

建议：如果要封装一个基类，最好将基类的析构函数声明为虚析构函数，以免后续的操作造成内存泄露。

8.7 纯虚函数和抽象类

在定义一个基类时，有时无法确定基类中虚函数的具体实现，那么就可以将虚函数声明纯虚函数。声明纯虚函数的格式:
virtual <函数类型> <函数名>(<形参表>) = 0;
抽象类：指至少包含一个纯虚函数的类，本身不能被实例化，称为抽象类。

8.7.1 纯虚函数的一些特性:

1. 含有纯虚函数的类(抽象类)，不能实例化对象;
2. 纯虚函数有没有定义函数体都是允许的;
3. 由抽象类派生出来的派生类，必须对基类中的纯虚函数重定义(重写)。

```
1  #include <iostream>
2  using namespace std;
3
4  class CShape
5  {
6  public:
7      virtual void Show() = 0;
8  };
9
10 class CTriangle : public CShape
11 {
12 public:
13     void Show()
14     {
15         cout << "CTriangle::Show()" << endl;
16     }
17 };
18
19 class CCirrcle : public CShape
20 {
21 public:
22     void Show()
23     {
24         cout << "CCirrcle::Show()" << endl;
25     }
26 };
27
28 int main()
```

```
{
    CShape *base1 = new CTriangle;
    CShape *base2 = new CCirrcle;
    base1->Show();
    base2->Show();
    return 0;
}
```

执行结果:

```
CTriangle::Show()
CCirrcle::Show()
```

8.8 纯虚析构函数

纯虚析构函数与纯虚函数有点区别:

8.8.1 必须为纯虚析构函数提供一个函数体;

8.8.2 由抽象类派生出来的派生类，可以不必重定义基类中的纯虚析构函数。

第1点的理由:

我们都知道，在类销毁进行析构的期间，会调用类的析构函数。如果不对一个纯虚析构函数进行定义，在析构函数就调用不到析构函数体了。

第2点的理由:

纯虚函数在派生类中必须要重新定义，但对于纯虚析构函数，在派生类中可以不必重定义？？原因是：编译器如果发现一个类当中没有显示声明构造函数或析构函数，编译器将会自动的生成一个默认版本，所以如果用户没有重定义纯虚析构函数，编译器会帮用户生成一个。但这里还得注意几个细节，得写个程序来调调才知道纯虚析构函数到底有多纯?

```
#include <iostream>
using namespace std;

class CAbstractBase
{
public:
    virtual ~CAbstractBase() = 0;
};

CAbstractBase::~CAbstractBase()
{
    cout << "请务必定义基类的纯虚析构函数" << endl;
}

class Derived : public CAbstractBase
{
};

int main()
{
    Derived d;
    return 0;
}
```

执行结果:

请务必定义基类的纯虚析构函数

举一反三：如果将基类中~CAbstractBase()的定义去掉，那么由CAbstractBase派生出来的Derived也将是抽象类，Derived d;声明就会失败。只有给基类的纯虚析构函数提供了函数体，Derived派生类才能够实例化对象。

8.9 虚函数其他的一些特性

8.9.1 C++的虚机制在构造函数和析构函数不工作

在类的构造函数，和非虚析构函数或许析构函数，C++的虚机制都不工作，就算调用了虚函数，也只能调用到本地版本。

```
#include <iostream>
using namespace std;

class CBase
{
public:
    CBase()
    {
        cout << "构造函数调用虚函数" << endl;
        func();
    }
    virtual void func()
```

```
    {
        cout << "本地版本:␣CBase::func()" << endl;
    }
    ~CBase()
    {
        cout << "析构函数调用虚函数" << endl;
        func();
    }
};

class CDerived : public CBase
{
public:
    void func()
    {
        cout << "重写版本:␣CDerived::func()" << endl;
    }
};

int main()
{
    CBase *base = new CDerived;
    cout << endl;
    delete base;
    return 0;
}
```

执行结果:

构造函数调用虚函数
本地版本: CBase::func()
析构函数调用虚函数
本地版本: CBase::func()

如果将基类的析构函数声明为虚析构函数，结果还是一样。

在博文，我只是稍微总结一下，这当然是非常浅的。不过我在学习C++面向对象这一部分，或者其他部分，都是结合好几本书的内容一起看，只有看我需要的那部分，这里的书包括：《C++程序设计语言特别版》，《C++ Primer中文版(第4版)》，《C++编程思想(第2版)第1卷标准C++引导》，《C++入门经典(第3版)》。我这些书全是电子版的，都是在大家论坛下载的，大家论坛的资源真是数不甚数。因为看书我不可能把一整本书全看，所以买书对我来说很浪费。再者每本书都有自己的优缺点，只能取其精华。

9 运算符重载

9.1 重载运算符

一般说来，c++内定义类型（例如，int,char,float等）的操作用运算符来表示，其调用形式是表达式。用户定义类型的操作则用函数来表示，对它采用显式调用。为了使用户定义的类型与内定义类型一致，也允许用户定义类型使用运算符来表示操作。这种一致性还表现在可以为用户定义类型提供初始化函数，赋值函数以及转换规则等。如果利用重载的运算符来增添新的类型，例如复数，集合等，这些用户定义的新类型可以象内部定义的类型一样来使用。对c++各种运算符的重新定义，既运算符的重载。

假设定义了一个复数类，它可以将两个复数相加，例如:

```
class comlpex{
     private:
        double epart; //复数的实部
        double ipart; //复数的虚部
     public :
        complex()
        {
               rpart+ipart+0.0; //缺省构造函数
        }
        complex(double rp,double ip)
        {
               rpart=rp;
               ipart=ip;
        }
       complex add(const complex&com)
       {
                 complex temp;
                 temp.rpart=com.rpart+rpart;
                 temp.ipart=com.ipart+ipart;
                 return temp;
       }
};
```

现在，就可以用它来进行两复数的加法运算了:

```
complex a(10,7),b(3,5);
complex c=a.add(b); //即a+b
```

这种加法运算采用函数调用的形式，并不直观。我们希望能和整数或浮点数一样，采用+表达式来实施运算，为此，需要重载+运算符函数。运算符函数的名字是operator关键字，再接着重载的运算符。他的返回值和参数与普遍函数相同。现在，可以重新定义复数的加法：

```
class complex
{
   //....
   public:
         complex operator+(const complex&com)
         {
         complex temp(rpart+com.rpart,ipart+com.ipart);
              return temp;
         }
  //...
};
```

运算符函数operator+被定义为公有的，以便程序的其他函数能使用它，在定义了函数之后，就可以象在内定义类型上一样，对复数对象用+表达式实施运算，例如：

```
complex 1(10,7),b(3,5),c;
c=a+b
```

c++编译器把表达式a+b解释为函数调用a.operator+(b).在调用它时，operator+成员函数首先创建一个临时的complex对象temp,然后把出现在加法表达式中的两个复数之和暂存其内，最后将这个临时对象返回.

9.1.1 重载运算符与重载一般函数的区别

区别主要是在参数的个数上，c++的运算符所能操作的操作数的个数的个数是规定好的。在重载二元运算符时只能指定两个参数，而重载一元运算符时只能指定一个参数。在重载函数时，可以指定任意的返回类型，甚至可以指定void类型作为返回类型，因为c++允许反调用函数而可以不使用函数的返回值。但c++的运算符是用在表达式中，一个运算符的运算结果要供别的运算符使用，因此任何运算符都指定有非void类型的返回类型。

9.1.2 类运算符与友员运算符的选取

一般而言，对于二元运算符，将它重载为一个友员运算符比重载为一个成员运算符要便于使用。作为一个友员函数，这个运算符不要求第一个参数一定为某个类的对象。一元运算符重载为一个成员函数最恰当，重载为友员也可以。但若将增1或减1运算符重载为友员运算符，则要使用引用参数。

注：赋值运算符不能重载为友员运算符.

9.2 用成员函数重载运算符

(1)运算符成员函数只能定义运算符的含义,不能改变运算符的优先级和结合顺序.例如,不论按何种方式重载运算符,a+b*c始终是先乘后加；而a=b=c也要先做b=c,然后,再对a赋值.

(2)运算符重载时,不能改变其目数.例如,任何企图把%定义为单目运算符,或把!定义为双目运算符的做法都会出错.

(3)运算符函数即可在类中定义,也可以在类外定义.在类外定义时,它至少应有一个相应类的参数.

(4)对于双目运算符@，当它用作成员函数时，只带一个参数（另一个运算符有调用它的对象给出），此时aa@bb与aa.operator@(bb)等价，当他用作外部函数时，带有两个参数（分别说明两个运算数），此时aa@bb与aa.operator@(aa,bb)等价

(5)对于单目前缀运算符@，当它用作成员函数时，不必带参数（运算符就是调用它的对象），此时@aa与aa.operator@()等价，当它用作外部函数时，带有一个参数（分别说明两个运算数），此时@aa与operator@(aa)等价

(6)对于单目后缀运算符@，当它用作成员函数时，应该带一个int参数（此参数一般不会使用，仅仅是为了把它同单目前缀运算符区分开），此时aa@与aa.operator@(int)等价，当它用作外部函数时，他应该带有两个参数，第一个是相应类的参数，第二个是int参数（也是用它来区分前后缀运算符），此时@aa与operator@(aa,int)等价.

(7)无任是在类中定义的运算符成员函数，还是在类外定义的运算符函数，都可以进行重载。也就是说可以定义多个同名的运算符函数。但其参数类型应有差别，否则会产生二义性。

(8)用户可重载已有的运算符，但不能定义自己的运算符，因为这常常会带来二义性，假设自定义**,以便进行乘方运算，编译器将无法确定它是按左结合（象fortran中那样），还是按有结合（象algol中一样）；也无法确定表达式a**b是被当作a*(*b)还是a**b.

(9)编译器预定义了=(赋值)，&(取地址)，(顺序)这三种运算符，不必重载就可以使用它们。如果要限制外部函数对它的使用，可以把它们声明为私有的.

9.3 用友员函数重载运算符

用类的友员函数重载运算符称为友员运算符。这种重载运算符函数的语义可表示为：operator<一元运算符>(<对象>)或operator<二元运算符>(<对象1>,<对象2>)。

前者可解释为:对当前对象进行一元运算;

后者可解释为:对当前对象1和对象2进行某二元运算.

虽然友员函数不是类的成员，但如果一个类声明它为友员，他就可以访问此类的任何私有或受保护的成员。就可以重载运算符

9.4 赋值运算符

在c和c++语言中，赋值被处理为表达式的一部分。赋值是一个二元运算符，其特别之处是其左操作数必须是左值。在自定义的类中可以重载赋值运算符。重载的赋值运算符与复制构造函数有许多共同特征：它们必须是类的成员函数，不允许是类的友员，它们不能被派生类继承。如果没有定义类的赋值运算符，在需要时，编译器会自动生成一个标准的赋值运算符。但赋值运算符可以有返回值，而构造函数不能。赋值运算符只有一个参数，如果参数是对自身类的引用，它将一个对象赋予另一个对象。其他类型的参数可以用来定义外来类型的赋值转换不。一般情况下，赋值运算符函数返回引用给被赋值的对象。

例如，在complex类中，作如下赋值：

```
complex c1,c2;
c1=c2;
```

编译器将c2.rpart拷贝到c1.rpart.并把c2.rpart拷贝到c1.rpart。在大多数情况下，它工作的很好，如果缺省的赋值操作不适用于我们写出的类，就应该重载=运算符，指定一个约定的赋值操作，例如，请看以下类CMessage,它被设计来存储和显示消息:

```
#include <iostream.h>
extern "c"{
#include<string.h>
}
class cmessage
{
        private:
              char *buffer;
        public:
              cmessage()
              {
                   buffer=new chat('\0');
              }
              ~cmessage()
              {
                   delete[]buffer;
               }
              void display()
              {
                   cout<<butter<<'\n';
              }
              void set(char *string)
              {
                   delete[]buffer;
                   buffer=new char[strlen(string)+1];
                   strcpy(buffer,string);
              }
};
```

现在对

```
cmessage c1,c2;
c1=c2;
```

在上例中。c1和c2个分配了一个字符数组c1.buffer和c2.buffer,由于缺省的赋值操作是按位拷贝，c1=c2使得c1.buffer中存放的地址被c2.buffer覆盖。结果，一方面由于没有指针指向原来c1.buffer所指的那块区域，这块存贮区域不能再被引用，而且一直要到程序结束时才释放，另一方面，c1对象和c2对象的数据成员指向的是同一存储块，如果以后为了改变放在一个对象内的消息而调用set()成员函数时，这个存储块就释放了，另一对象因buffer成员释放则指向难以预料的数据。（若一对象在另一对象之先撤消也会出项同样情况，因为析构函数也会释放buffer指向的存储块。）

为提供一个cmessage对象向另一个赋值的合适的例程，则在类定义中增加了以下运算符函数:

```
class cmessage
{
         //其他声明
         public:
             void operator=(const cmessage&message)
             {
                   delete[]buffer;
                   buffer=new char[strlen(message.buffer)+1];
                   strcpy(buffer,message.buffer);
             }
         //其他声明...

};
```

这个例程不仅仅是将存储块的地址（buffer所指）从源对象复制到目的的对象，重载的=运算符函数为目的对象创建了一个新存储块，再把消息串拷贝到其中，于是，每个对象都有自己的串拷贝。

注意：重载=运算符必须用成员函数，不得用非成员的友员函数。

9.5 下标运算符重载

我们常用下标运算符operator[]来访问数组中的某个元素.它是一个双目运算符,第一个运算符是数组名,第二个运算符是数组下标.在类对象中,我们可以重载下标运算符,用它来定义相应对象的下标运算.注意,C++不允许把下标运算符函数作为外部函数来定义,它只能是非静态的成员函数.下标运算符定义的一般形式:

```
T1 T::operator[](T2; )
```

其中,T是定义下标运算符的类,它不必是常量.T2表示下标,它可以是任意类型,如整形,字符型或某个类.T1是数组运算的结果.它也可以是任意类型,但为了能对数组赋值,一般将其声明为引用形式.在有了上面的定义之后,可以采用下面两种形式之任一来调用它:

```
x[y]
```

或

```
x.operator[](y)
```

x的类型为T,y的类型为T2.

下面来看一个简单的例子:

```
#include<iosteram.h>
extern"C"{
#include<stdlib.h>
}
class ainteger{
        int *a;
        int sz;
   public:
        ainteger(int size)
        {
            sz=size;
            a=new int[size];
        }
        int &operator[](int i)
        {
             if(i<0||i>=sz) //越界
             {
                    cout<<"error"<<endl;
                    exit(1);
              }
             return a[i];
        }
        ~ainteger()
        {
                delete[]a;
        }
};
```

在整形数组ainteger中定义了下标运算符，这种下标运算符，这种下标运算符能检查越界的错误。现在使用它:

```
ainteger ai(10);
ai[2]=3;
int i=ai[2];
```

对于ai^2=3,他调用ai.operator(2),返回对ai::a^2的引用，接着再调用缺省的赋值的赋值运算符，把3的值赋给此引用，因而ai::a^2的值为3。注意，假如返回值不采用引用 形式，ai.operator(2)的返回值是一临时变量，不能作为左值，因而，上述赋值会出错。对于初始化i=ai^2,先调用ai.operator(2)取出ai::a^2的值。然后再利用缺省的复制构造函数来初始化i.

9.6 重载减1增1运算符

前缀增（减）量和后缀增（减）量都用++（–）运算符表示，并且都是单目运算符。为了区分它们，编译器规定，后缀增减量应带有一整形参数，这种参量仅仅是用来分辨前后缀。调用时，不必显示地使用此参数，它的缺省值为零。增1减1运算符(++,–)可被重载为前缀或后缀运算符，也可以被重载为类运算符或友员运算符。对类x的对象a,下表列出其增量和减量的定义形式和调用形式:

运算符名	成员函数定义形式	外部函数定义形式	调用形式
前缀++	X operator++()	X operator++(X&)	++a
后缀++	X operator++(int)	X operator++(X&,int)	a++
前缀–	X operator–()	X operator–(X&)	–a
前缀–	X operator–(int)	X operator–(X&,int)	a–

对于内部类型前缀++（–）和后缀++（–）的含义是不同的，++a中表达式的值和a的值都为原来a的值加一，a++中表达式的值为原来的a,a的值为原来的a加一。

在运算符函数中，表达式的值是由运算符函数的返回值给出，对象的值则由对象成员的值来反映。对于++（–）运算符，尽管它提供了（后缀形式的增（减）量在AT&T3.0版及更高版本中才实现）前缀和后缀两种形式，但由于返回值和对象值都有用户指定，用户定义的++（–）运算符前后缀的语义可以与内部定义的类型不一样。

9.7 类型转换运算符

类型转换是c语言中的一个关键特征，c++提供两种方法来自动地处理类型转换。第一种是建立类的构造函数。转换构造函数可以将外来类型转换成类的一个对象。第二中是建立转换运算符函数，它可以将一个对象自身的类型转换为其他类型。定义一个转换函数使用关键字operator,后跟要转换的类型的名字。转换函数有以下限制:

1.转换函数不能带任何参数，并且总是成员函数。

2.不能说明类型转换函数的返回类型。隐含的返回类型是要转换的类型。在函数体中，适当的值必须返回。转换函数也可以通过显示类型转换表达式被调用，或使用名字直接调用。

[2]DEFINITION NOT FOUND.

10 C++ I/O流库

10.1 流和流库

10.1.1 概 述

在C语言中，输入／输出系统的特点是缺乏类型检查机制。如printf函数，在格式控制字符串后的参数，即使类型和个数与其不匹配，编译是不会出错，但运行时会得到错误的结果。C＋＋提供了新的输入／输出方式。其主要目标是建立一个类型安全，扩展性好的输入／输出系统。在一个类型安全的输入／输出系统中，类似上述printf的错误在编译时就可发现。一个理想的可扩展的输入／输出系统必须能以两种方式进行扩展：

1.能够包含用户定义的数据类型。

2.能够包含新的输入／输出方法。

在C＋＋中输入／输出流库充分利用了C＋＋的面向对象的特性实现了上述目标。

10.1.2 流的概念

所谓流是指数据从一个位置流向另一个位置。流是C＋＋为输入／输出提供的一组类，都放在流库中。流总是与某一设备相联系（例如，键盘，屏幕或硬盘等），通过使用流类中定义的方法，就可以完成对这些设备的输入／输出操作。一般，若要在流中存储数据，这个流为输出流；要从流中读取数据，这个流为输入流。有的流既是输入流，又是输出流。流类形成的层次结构就构成流类库，即流库。与C 语言中的输入／输出流库一样，C++的输入输出流库不是语言的一部分，而是作为一个独立的函数库提供的。因此，在使用时需要包含相应的头文件。

输入流和输出流:在编写程序时,常要输入一些数据,在处理完数据之后,有要把结果输出. c++ 没有专门的输入输出语句,输入输出都有流库来处理.通过输出流,拥护可以从这些设备中读取数据;通过输出流则可以往设备中写数据.

输出流:我们用cout输出过数据.实质上.cout 就是输出流类ostream的派生类预定义的一个对象.它与标准输出设备相联系,以便把数据送往屏幕显示.在ostream类中,重载了«运算符,用来处理各种内部类型的输出

输入流:c++也为输入定义了一个流类istream.这个类中重载了»运算符,以便从先观的设备中读取数据,对应与插入运算.这里»运算符内称为析取运算.

10.2 流运算符

10.2.1 重载插入运算符

在ostream类中，重载了«运算符，用来处理各种内部类型的输出，ostream的定义放在iostream.h头文件中：

```
class ostream:public virtual {ios
            //......
            public:
                ostream&operator<<(const char; *)
                ostream&operator<<(char; )
                ostream&operator<<(short i){
                      return *this<<int(i);
                }
                ostream&operator<<(int);
                ostream&operator<<(long);
                ostream&operator<<(double);
                ostream&operator<<(const void*);
           // ......}
;
```

注：其中char*用于输出字符串，void*用于输出指针的地址值

10.2.2 重载析取运算符

```
class istream:public virtual {ios
                 //......
                public:
                    istream&operator>>(char ; *)//字符串
                    istream&operator>>(char ; &)//字符
                    istream&operator>>(shirt ; &)
                    istream&operator>>(int ; &)
                    istream&operator>>(long ; &)
                    istream&operator>>(float ; &)
                    istream&operator>>(double ; &)
                 //...
};
```

istream将ios作为基类.他也为所有内部类型定义了析取运算.在iostream.h头文件总预定义了一个istream派生类的对象cin,cin与标准输入相联系,用语从键盘输入数据.

例如:

```
//...
int i;
cin >>i;
//...
```

cin»i将调用cin.operator.(i),把键盘中输入的整数存放到变量i中.

10.3 格式控制

在前面，输入／输出的数据没有指定格式，它们都按缺省的格式输入／输出。然而，有时需要对数据格式进行控制。这时需利用ios类中定义的格式控制成员函数，通过调用它们来完成格式的设置。ios类的格式控制函数如下所示:

10.3.1 宽度控制

ios的成员函数width()是指定在输入/输出一个数字或串时,缓冲去可存储的最大字符数,在输入流总,长用他来防止缓冲区溢出,
例如:

```
char buffer[20];
cin.width(20);
cin>>buffer;
```

width()调用告诉cin插入操作一次最多能读入20个字符,从而可以保证buffer中的数不会因为超过20而溢出.

10.3.2 格式状态

在ios类中,定义了一个表示流状态的枚举,枚举中有各种标志,用它们进一步控制输入输出,
在ios类中定义了几个成员函数,用来设置,读取和取消标志位:
long flags( ) const 返回当前的格式标志。
long flays(long newflag) 设置格式标志为newflag，返回旧的格式标志。
long setf(long bits) 设置指定的格式标志位，返回旧的格式标志。
long setf(long bits,long field)将field指定的格式标志位置为bits，返回旧的格式标志
long unsetf(long bits) 清除bits指定的格式标志位，返回旧的格式标志。
long fill(char c) 设置填充字符，缺省条件下是空格。
char fill( ) 返回当前填充字符。
int precision(int val) 设置精确度为val，控制输出浮点数的有效位，返回旧值。
int precision( ) 返回旧的精确度值。
int width(int val) 设置显示数据的宽度(域宽),返回旧的域宽。
int width( )只返回当前域宽，缺省宽度为0。这时插入操作能按表示数据的最小宽度显示数据

10.3.3 控制符

在前面.控制输入/输出采用的是函数的形式,但他们使用起来并不方便.为此,c++提供了控制符,他可以直接插入或析取运算中.
预定义的操纵算子
使用成员函数控制格式化输入输出时，每个函数调用需要写一条语句，尤其是它不能用在插入或提取运算符的表达式中，而使用操纵算子，则可以在插入和提取运算符的表达式中控制格式化输入和输出。在程序中使用操纵算字必须嵌入头文件iomanip.h
一般,预定义的控制符有以下几种:
dec 十进制的输入输出
hex 十六进制的输入输出
oct 八进制的输入输出
ws 提取空白字符
ends 输出一个nul字符
endl 输出一个换行字符，同时刷新流
flush 刷新流
resetiosflags(long) 请除特定的格式标志位
setiosflags(long) 设置特定的格式标志位
setfill(char) 设置填充字符
setprecision(int) 设置输出浮点数的精确度
setw(int) 设置域宽格式变量

10.4 其它流函数

在ios,istream和ostream类中,还定义了若干输入输出函数,它们主要用于错误处理,流的刷新以及流输入输出方式的控制.
错误处理
在对一个流对象进行I/O操作时，可能会产生错误。当错误发生时，错误的性质被记录在ios类的一个数据成员中。
ios类中定义的描述错误状态的常量:
goodbit————-没有错误，正常状态
eofbit————到达流的结尾
failbit————-I/O操作失败，清除状态字后，可以对流继续进行操作。
badbit————试图进行非法操作，清除状态字后，流可能还可以使用。
hardfail————致命错误，不可恢复的错误。
对应于这些位,可用ios中定义的如下函数来检查流的当前状态位:
int good()———如果正常,返回非0值
int bad()———-如果badbit被设置,返回非0值
int eof()———-如果eofbit被设置,返回非0值
int fail()———如果failbit被设置,返回非0值
int rdstate()——返回当前错误状态位.
流的其它成员函数可以从流中读取字符或字符串，对流进行无格式化的输入 输出操作，以及直接控制对流的I/O操作。

返回类型	ostream类的成员	描　　述
ostream&	put(char ch)	向流中输出一个字符ch，不进行任何转换
ostream&	write(char*,int)	向流中输出指定长度的字符串，不进行转换
ostream&	flush( )	刷新流，输出所有缓冲的但还未输出的数据
ostream&	seekp(streampos)	移动流的当前指针到给定的绝对位置
ostream&	seekp(sereamoff,sek_{dir})	流的当前指针类似与文件的当前指针
streampos	teelp( )	返回流的当前指针的绝对位置

istream类的成员函数

返回类型	istream类的成员	描　　述
int	get( )	读取并返回一个字符
istream&	get(char&c)	读取字符并存入c中
istream&	putback( )	将最近读取的字符放回流中
istream&	read(char*,int)	读取规定长度的字符串到缓冲区中
int	peek( )	返回流中下一个字符，但不移动文件指针
istream&	seekg(streampos)	移动当前指针到一绝对地址
istream&	seekg(streampos,seek_{dir})	移动当前指针到一相对地址
streampos	tellg( )	返回当前指针

10.5　文件流

C++系统通过对流类进一步扩展，提供了支持文件I/O的能力，使得程序员在建立和使用文件时，就像使用cin和cout一样方便。下图新派生的五个类用于文件处理。fstreambase类提供了文件处理所需的全部成员函数，在它的派生类中没有提供新的成员函数。 ifstream类用于文件的输入操作；ofstream类用于文件的输出操作，fstream类允许对文件进行输入/输出操作。这几个类同时继承了前面介绍的流类的基本类等级中定义的成员函数。使用这些类时，必须在程序中嵌入头文件fstream.h通过打开一个文件，可将一个流与一个文件相联结。

filedbuf是streambuf的派生类，提供对文件缓冲区的管理能力。我们一般不涉及这个类C++系统通过对流类进一步扩展，提供了支持文件I/O的能力，这使得程序员在建立和使用文件时，就像使用cin和cout一样方便。左图新派生的五个类用于文件处理。fstreambase类提供了文件处理所需的全部成员函数，在它的派生类中没有提供新的成员函数。ifstream类用于文件的输入操作；ofstream类用于文件的输出操作，fstream类允许对文件进行输入/输出操作。这几个类同时继承了前面介绍的流类的基本类等级中定义的成员函数。使用这些类时，必须在程序中嵌入头文件fstream.h通过打开一个文件，可将一个流与一个文件相联结。在ios类中定义的一组枚举常量名给出了可允许的文件打开方式:

in———————— 打开一个文件进行操作
out———————- 打开一个文件进行写操作
ate———————- 文件打开时将文件指针指向文件尾
app———————- 添加，输出的内容添加到文件尾
trunc———————— 若文件存在，清除原有内容，将长度截为0
nocreat———————— 若文件不存在，打开操作失败

10.6　文件的打开与关闭

10.6.1　在C++中，打开一个文件就是将这个文件与一个流建立关联，关闭一个文件就是取消这种关联。要执行文件的输入/输出，必须做三件事:

1. 在程序中包含头文件ifstream.h
2. 建立流。建立流的过程就是定义流类的对象，例如:

```
ifstream ；定义了输入流对象inin ；定义了输出流对象
ofstreamoutout ；定义了输入
fstreamio输出流对象/io
```

3. 使用open( )函数打开文件，是某一文件与上面的某一流相联系。

open( )函数的原形为void open(const unsigned char*,int mod,int $\text{file}_{\text{attrb}}$)；文件属性。取值为0代表普遍文件，1为隐藏文件等，缺省值为0。打开的文件使用完毕，必须将它关闭，关闭文件使用close( )函数，close( )函数也是流类中的成员函数。

打开一输入文件

```
ifstream input;
input.open("test1",ios::in,0);
```

打开一输出文件

```
ofstream output;
output.open("test2",ios::out,0);
```

打开一输入／输出文件

```
fstream both;
both.open("test",ios::in,ios::out,0);
```

可简化为:

```
fstream both("test",ios::in,ios::out,0);
```

关闭文件:

```
both.close( );
```

文件流成员函数open()的格式如下所示:

```
void open(char * name,int mode,int file_attrb);
```

在这里name表示文件名，mode是下列值之一:
ios::app————————所有数据以附加方式写到流
ios::ate————————打开文件，并把文件指针移到文件尾
ios::in————————为读打开文件
ios::out————————为写打开文件
ios::trunc————————如果文件存在，舍去文件内容
ios::nocreate————————如果文件不存在，则失败
ios::noreplace————————如果文件存在，则失败

10.7 文件的读写

C++把文件看作是字符序列，即所谓流式文件。根据数据的组织形式，文件又可分为ASCII码文件和二进制文件两种。如果要进行文件的输入输出，必须先建立一个流，然后将这个流与文件相关联，即打开文件，此时才能进行读写操作，完成后再关闭这个文件。如果建立的文件是供人阅读的，它必须以文本方式打开，如果是供其它程序使用，则可以使用二进制文件或文本文件。

10.8 另一个源头来的

在C++输入输出流控制中，就把话语权交给iomanip吧。
以下列出一些比较常用的设置方法:
包含头文件：#include <iomanip>
dec 十进制 dec(c++) == %d(c)
hex 十六进制 oct(c++) == %o(c)
oct 八进制 hex(c++) == %x(c)
setfill(c) 填充字符为c
setprecision(n) 设置n个有效数字
setw(n) 设输出的宽度为n
setiosflags(ios::fixed) 固定输出小数点个数
setiosflags(ios::scientific) 输出指数
setiosflags(ios::left) 左对齐
setiosflags(ios::right) 右对齐
setiosflags(ios::skipws 忽略前导空白
setiosflags(ios::uppercase) 16进制数大写输出
setiosflags(ios::lowercase) 16进制小写输出
setiosflags(ios::showpoint) 强制显示小数点
setiosflags(ios::showpos) 强制显示符号(+/-)
在设置mask有两个方法：setiosflags和setf。
这些用法看头文件就很清楚了，主要在：ios_{base}.h和iomanip中。