第二章　类与函数

本章导读

类与函数关系密切。在C++中，函数的位置可以在类之外(常规函数)，也可以在类之内(成员函数)，还可以在类之外，但具备类之内的特征(友元函数)。运算符重载本质上的函数，是函数的重载，但是不能违背运算本身所具有的特征。

另外，在C++中经常用const关键字和&符号修饰函数，对提高函数质量、提升执行效率、完善工程管理都具有重要意义。

学习目标：

1. 认识类生存期相关函数(构造函数、拷贝构造函数、析构函数)；
2. 认识类与函数的关系；
3. 认识运算符重载；
4. 认识const关键与&符号对函数的意义；
5. 在类中使用new/delete运算符；

本章目录

第一节　构造函数

构造函数(Constructor)是与类名称相同的一种特殊函数，是对象从无到有的过程中所必须执行的函数，是可以显式存在也可以隐式存在的一种函数。当一个类中的构造函数显式存在时，可以有一个，也可以有多个构造函数。例程2-1有隐式构造函数，例程2-2有显示构造函数。

例程2-1

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  class numUpDown{
第5行  public:
第6行      void setNum(int UP,int DOWN){
第7行          this->upNum=UP;
第8行          this->downNum=DOWN;
第9行      }
第10行     void print(){
第11行         std::cout<<this->upNum<<"/"<<this->downNum;
第12行     }
```

```
第13行  private:
第14行      int upNum;
第15行      int downNum;
第16行  };
第17行
第18行  int main(){
第19行      numUpDown UD;
第20行      UD.setNum(1,2);
第21行      UD.print();
第22行  }
```

在例程2-1中，没有与类名相同的函数存在。在源代码编译过程中，会自动生成构造函数，这就是隐式构造函数。对开发人员而言，该构造函数似乎并不存在，但在对象生成过程中，仍然将调用自动形成的构造函数。在执行例程第19行numUpDown UD时，将自动调用隐式构造函数。隐式构造函数也称默认构造函数。

在例程2-2中，有两个与类名相同的函数存在，此即为显式构造函数。当类中含有显式构造函数时，隐式构造函数将不允许存在，如numUpDown UD0将可能调用隐式构造函数(例程2-1所示)或无参构造函数(如例程2-2所示)，但当有显示构造函数且没有无参构造函数时，执行numUpDown UD0将出现错误。

例程2-2

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   class numUpDown{
第5行   public:
第6行       numUpDown(){
第7行           this->upNum=1;
第8行           this->downNum=1;
第9行       }
第10行      numUpDown(int Up,int Down){
第11行          this->upNum=Up;
第12行          this->downNum=Down;
第13行      };
第14行      void setNum(int UP,int DOWN){
第15行          this->upNum=UP;
第16行          this->downNum=DOWN;
第17行      }
第18行      void print(){
第19行          std::cout<<this->upNum<<"/"<<this->downNum;
第20行      }
第21行  private:
第22行      int upNum;
第23行      int downNum;
第24行  };
第25行
第26行  int main(){
第27行      numUpDown UD0;//调用：numUpDown()构造函数
```

```
第28行      //如果没有无参构造函数，将不能按上述语句申明对象
第29行      UD0.print();//输出:1/1
第30行
第31行      numUpDown UD1(1,2);//调用：numUpDown(int Up,int Down)
第32行      UD1.print();//输出：1/2
第33行  }
```

注意：当调用隐式构造函数或者调用无参构造函数时，对象名称后不能有括号，如numUpDown UD0()，果如此，编译器将示UD0()为函数申明。另外，例程2-2可以优化如例程2-3。

例程2-3

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  class numUpDown{
第5行  public:
第6行      numUpDown(int Up=1,int Down=1){
第7行          this->upNum=Up;
第8行          this->downNum=Down;
第9行      };
第10行     void setNum(int UP,int DOWN){
第11行         this->upNum=UP;
第12行         this->downNum=DOWN;
第13行     }
第14行     void print(){
第15行         std::cout<<this->upNum<<"/"<<this->downNum;
第16行     }
第17行 private:
第18行     int upNum;
第19行     int downNum;
第20行 };
第21行
第22行 int main(){
第23行     numUpDown UD0;//调用：numUpDown(int Up,int Down)全部采用默认值
第24行     UD0.print();//输出:1/1
第25行
第26行     numUpDown UD1(2,3);//调用：numUpDown(int Up,int Down)，全部不采用默认值
第27行     UD1.print();//输出：2/3
第28行
第29行     numUpDown UD2(2);//调用：numUpDown(int Up,int Down)，第二个参数采用默认值
第30行     UD2.print();//输出：2/1
第31行 }
```

第二节　析构函数

析构函数是对象灭失时自动执行的函数，函数名称为在类名称前加1个~符号，如例程2-4所示第10行~numUpDown()所示，其功能是在对象灭失前做最后的善后工作。和构造函数类似，析构函数也分为隐式和显式两种，例程2-4是显式析构函数，当类中没有显示构造函数

时，在编译过程中，将自动增加隐式析构函数。和构造函数不同，一个类可以有1个或多个构造函数，但一个类有且仅有一个析构函数。

例程2-4

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  class numUpDown{
第5行  public:
第6行      numUpDown(int Up=1,int Down=1){
第7行          static int initCount=0;//调用构造函数次数
第8行          ++initCount;
第9行          this->upNum=Up;
第10行         this->downNum=Down;
第11行         this->id=initCount;
第12行     };
第13行     ~numUpDown(){
第14行         std::cout<<"Now Exit numUpDown Object!!";
第15行         std::cout<<"numUpDown ID:"<<this->id<<std::endl;
第16行     }
第17行     void setNum(int UP,int DOWN){
第18行         this->upNum=UP;
第19行         this->downNum=DOWN;
第20行     }
第21行     void print(){
第22行         std::cout<<this->upNum<<"/"<<this->downNum;

第23行     }
第24行 private:
第25行     int id;
第26行     int upNum;
第27行     int downNum;
第28行 };
第29行
第30行 numUpDown rtnUD(){
第31行     numUpDown UD;//调用构造函数
第32行     std::cout<<"Next Executing 'return UD;'\n";
第33行     return UD;
第34行 }
第35行 int main(){
第36行     numUpDown UD0;//调用：numUpDown(int Up,int Down)全部采用默认值
第37行     UD0.print();//输出:1/1
第38行
第39行     numUpDown UD1(2,3);//调用：numUpDown(int Up,int Down)，全部不采用默认值
第40行     UD1.print();//输出：2/3
第41行
第42行     numUpDown UD2(2);//调用：numUpDown(int Up,int Down)，第二个参数采用默认值
```

```
第43行      UD2.print();//输出：2/1
第44行
第45行      rtnUD();//调用rtnUD()函数
第46行  }
```

```
1/12/32/1Next Executing 'return UD;'
Now Exit numUpDown Object!!numUpDown ID:4
Now Exit numUpDown Object!!numUpDown ID:4
Now Exit numUpDown Object!!numUpDown ID:3
Now Exit numUpDown Object!!numUpDown ID:2
Now Exit numUpDown Object!!numUpDown ID:1
```

图2-1　例程2-4执行效果图

在例程2-4中，当执行第45行代码rtnUD();时，函数rtnUD()将首先生成一个局部对象UD(第31行)，然后执行第32行，最后再返回局部对象UD，UD消失，执行析构函数中第14-15行代码。

观察图deConstructor00会发现有两个“4”出现，根据第25行“int id”定义以及第7-11行代码的含义可知id值随着构造函数执行此处而变化，每个numUpDown类对象具有唯一的id值。但图中可知，id值为4的numUpDown类对象被销毁了两次，其原因是：第45行代码被执行后将调用第30-34行代码，在第31行中，将调用构造函数，此时id编号被设置为4(UD0、UD1、UD2编号顺次为1-3)，当rtnUD()函数执行完毕后，将灭失id为4的局部对象UD。函数有返回值，rtnUD()返回值为id为4的numUpDown类对象，该返回值存在周期极短，随即灭失，此时仍然会调用析构函数，因此会有两个4出现。

例程2-5更能体现析构函数的作用，其功能是在堆中动态开辟连续内存空间，当对象Arr灭失时，应该释放其开辟的内存空间，否则将导致内存泄露。在类中，释放内存空间的功能在析构函数中实现，即当Arr类的对象灭失时，调用析构函数释放内存空间。注意：new动态开辟的内存空间，并不能自动释放内存空间。

例程2-5

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class Arr{
第4行  public:
第5行      Arr(int size=10){
第6行          this->size=size;
第7行          this->ptr=new int[size];//开辟内存空间
第8行      }
第9行      void initNum(){
第10行         int count=this->size;
第11行         for(int i=0;i<count;++i)*(ptr+i)=i;//赋值与i变化相关
第12行     }
第13行     void print(){
第14行         int count=this->size;
第15行         for(int i=0;i<count;++i)std::cout<<*(ptr+i)<<" ";
第16行     }
第17行     ~Arr(){//析构函数，释放内存空间
第18行         delete []this->ptr;
第19行     }
第20行 private:
第21行     int size;
第22行     int* ptr;
第23行 };
第24行 int main(){
第25行     Arr myArr(25);
```

```
第26行      myArr.initNum();
第27行      myArr.print();
第28行
第29行      return 0;
第30行  }
```

第三节　拷贝构造函数

当依照已经存在的对象创建新的对象时，将调用类的拷贝构造函数。拷贝构造函数可以分为隐式和显式两种，当编程者自己撰写拷贝构造函数时，称之为显式拷贝构造函数；当类没有显式拷贝构造，由编译器根据需要自动生成一个隐式拷贝构造函数。

拷贝构造函数的名称用户类型相同，其参数为类类型引用对象。加入某类名称为clsName，其拷贝构造函数的格式为：clsName(clsName &objName)，其中objName为对象名称，与普通变量命名规则相同，如例程2-6第17-19行。

例程2-6

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   class numUpDown{
第5行   public:
第6行       numUpDown(int Up=1,int Down=1){
第7行           static int initCount=0;//调用构造函数次数
第8行           ++initCount;
第9行           this->upNum=Up;
第10行          this->downNum=Down;
第11行          this->id=initCount;
第12行      };
第13行      ~numUpDown(){
第14行          std::cout<<"Now Exit numUpDown Object!!";
第15行          std::cout<<"numUpDown ID:"<<this->id<<std::endl;
第16行      }
第17行      numUpDown(numUpDown &UD){
第18行          std::cout<<"Copy Constructor!!"<<endl;
第19行      }
第20行      void setNum(int UP,int DOWN){
第21行          this->upNum=UP;
第22行          this->downNum=DOWN;
第23行      }
第24行      void print(){
第25行          std::cout<<this->upNum<<"/"<<this->downNum<<std::endl;
第26行      }
第27行  private:
第28行      int id;
第29行      int upNum;
第30行      int downNum;
第31行  };
第32行
```

```
第33行  numUpDown rtnUD(numUpDown &UD){
第34行      std::cout<<"Print input UD:";
第35行      UD.print();
第36行      std::cout<<"Next Executing 'return UD;'\n";
第37行      return UD;
第38行  }
第39行  int main(){
第40行      numUpDown UD0;//调用：numUpDown(int Up,int Down)全部采用默认值
第41行      UD0.print();//输出:1/1
第42行
第43行      numUpDown UD1(2,3);//调用：numUpDown(int Up,int Down)，全部不采用默认值
第44行      UD1.print();//输出：2/3
第45行
第46行      numUpDown UD2(2);//调用：numUpDown(int Up,int Down)，第二个参数采用默认值
第47行      UD2.print();//输出：2/1
第48行
第49行      numUpDown UD3=rtnUD(UD0);//调用rtnUD()函数
第50行      UD3.print();
第51行
第52行      return 0;
第53行  }
```

```
1/1
2/3
2/1
Print input UD:1/1
Next Executing 'return UD;'
Copy Constructor!!
-858993460/-858993460
Now Exit numUpDown Object!!numUpDown ID:-858993460
Now Exit numUpDown Object!!numUpDown ID:3
Now Exit numUpDown Object!!numUpDown ID:2
Now Exit numUpDown Object!!numUpDown ID:1
```

图2-2　例程2-6执行效果图

当执行例程2-6第49行时，首先执行其中的rtnUD(UD0)，然后进入第33-38行，首先执行第34行后输出后紧接着输出实参UD0的内容，然后执行第36行结束后返回UD值(即实参UD0的值)。

在将临时变量赋值给UD3时，将调用拷贝构造函数，执行第17-19行代码。当执行50行代码时，输出错误，如图中到倒数第5行所示，其原因是拷贝构造函数不正确，其函数修改如例程2-7所示，其执行界面如图2-3。

例程2-7

```
第1行  numUpDown(numUpDown &UD){
第2行      std::cout<<"Copy Constructor!!"<<endl;
第3行      this->downNum=UD.downNum;
第4行      this->upNum=UD.upNum;
第5行      this->id=UD.id;
第6行  }
```

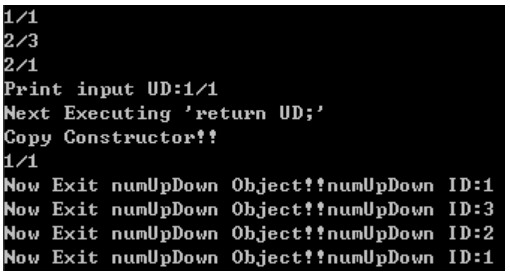

图2-3　拷贝构造函数修改后执行效果图

假定将rtnUD()函数的修改如例程2-8所示，其执行改变为图2-4所示。

例程2-8

```
numUpDown rtnUD(numUpDown UD){//删除&符号
    std::cout<<"Print input UD:";
    UD.print();
    std::cout<<"Next Executing 'return UD;'\n";
    return UD;
}
```

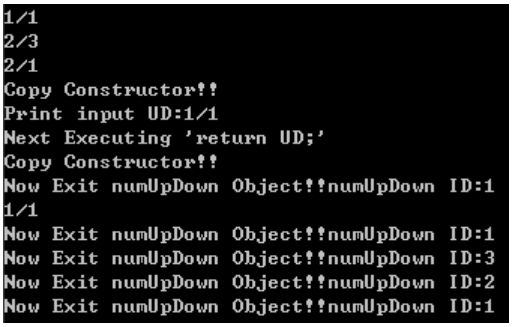

图2-4　拷贝构造函数修改后执行效果图

从图2-4中可以看出多执行了1次拷贝构造函数和析构函数，原因是当参数变为numUpDown rtnUD(numUpDown UD)后，参数传递变为传值，将执行一次复制，即函数将复制UD0并成为函数的局部对象变量，并函数执行完毕后，将析构该变量，因此将多执行1次拷贝构造函数和析构函数。

当类没有拷贝构造函数时，编译器会根据需要自动创建拷贝构造函数，简单而快速地将每个数据成员拷贝到新的对象中。当从拷贝构造函数创建新的对象时，无需执行构造函数，一般说来将提高效率。但是，默认的拷贝构造函数只能简单复制数据，当这样的工作不能满足需要时，就需要编程者创建显式拷贝构造函数，如指针地址等。当数据成员含有指针数据成员时，当执行默认构造函数时，其指针仍然指向源对象数据成员地址，包括该指针数据成员，如例程2-9所示。

例程2-9

```
#include<iostream>
using namespace std;
class Arr{
public:
    Arr(int size=10){
        this->size=size;
        this->ptr=new int[size];//开辟内存空间
    }
    void initNum(int Num=1){
        int count=this->size;
        for(int i=0;i<count;++i)*(ptr+i)=i*Num;//赋值与i变化相关
    }
    void print(){
```

```
第14行            int count=this->size;
第15行            for(int i=0;i<count;++i)std::cout<<*(ptr+i)<<" ";
第16行        }
第17行        ~Arr(){//析构函数，释放内存空间
第18行            delete []this->ptr;
第19行        }
第20行    private:
第21行        int size;
第22行        int* ptr;
第23行    };
第24行    int main(){
第25行        Arr myArr(25);
第26行        myArr.initNum(5);
第27行        std::cout<<"myArr输出：";
第28行        myArr.print();
第29行        std::cout<<std::endl;
第30行
第31行        Arr myArr2=myArr;
第32行        std::cout<<"myArr2输出：";
第33行        myArr2.print();
第34行        std::cout<<std::endl;
第35行
第36行        myArr2.initNum(3);
第37行        std::cout<<"执行initNum(3)之后myArr2输出：";
第38行        myArr2.print();
第39行        std::cout<<std::endl;
第40行
第41行        std::cout<<"myArr输出：";
第42行        myArr.print();
第43行        std::cout<<std::endl;
第44行
第45行        system("pause");
第46行        return 0;
第47行    }
```

```
myArr输出：0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110
 115 120
myArr2输出：0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 11
0 115 120
执行initNum(3)之后myArr2输出：0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51
54 57 60 63 66 69 72
myArr输出：0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 7
2
```

图2-5　例程2-9执行效果图

从图中可以看出，当执行拷贝构造函数后(第31行Arr myArr2=myArr)后第33行myArr2的输出与myArr输出一致。当执行myArr2.initNum(3)后，第38行myArr2.print()与第42行myArr.print()输出完全一致。其原因是ptr仍然指向相同的内存块。myArr2与myArr共享同一内存区域，所以针对其内存的改变，结果一致。值得注意的是：当程序运行结束时，将出现错误，原因是析构函数中delete []this->ptr语句将释放同一内存块两次。例程2-9增加拷贝构造函数如例程2-10所示，其执行结果如图2-6所示。

例程2-10

```
第1行  Arr(Arr &fromArr){
第2行      this->size=fromArr.size;
第3行      this->ptr=new int[this->size];
第4行  }
```

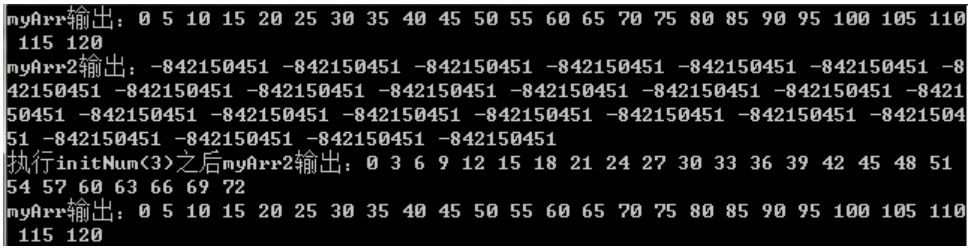

图2-6　例程2-9修改后执行效果图

图2-6中有一系列-842150451，其原因是执行拷贝构造函数中的this->ptr=new int[this->size]代码后，仅仅分配地址空间，并没有初始化，因此其值不确定(不同的机器不同状态可能不同)。当执行myArr2.initNum(3)后，内存空间存储其值被确定，因此输出相关内容。另外，从输出可以看出，myArr和myArr2的输出不同，其变化实现独立。

第四节　友元函数

一个类的私有成员不能被外部程序访问，但有时，允许一些可信函数访问一个类的私有成员，会使编程更加方便。C++允许使用关键字friend申明友元函数以访问类的私有成员。友元函数虽然是类的非成员函数却能够访问类的所有成员，类授予友元函数特别访问权。通常同一个开发者会出于技术和非技术的原因，控制友元函数的数量(如：当你想更新类时，还要征得或知晓友元函数开发者的同意)。

例程2-11

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  class numUpDown{
第5行  public:
第6行      numUpDown(int Up=1,int Down=1){
第7行          this->upNum=Up;
第8行          this->downNum=Down;
第9行      };
第10行     friend numUpDown Plus(numUpDown lUD,numUpDown rUD);//删除该行，会发生什么？
第11行     void print(){
第12行         cout<<this->upNum<<"/"<<this->downNum<<endl;
第13行     }
第14行 private:
第15行     int upNum;
第16行     int downNum;
第17行 };
第18行 numUpDown Plus(numUpDown lUD,numUpDown rUD){
第19行     return numUpDown(lUD.upNum*rUD.downNum+rUD.upNum*lUD.downNum,lUD.downNum*rUD.downNum);
第20行 }
第21行 int main(){
第22行     numUpDown UD0(2,3);
第23行     numUpDown UD1=Plus(numUpDown(2,5),UD0);
第24行     UD1.print();
第25行
```

```
第26行      system("pause");
第27行      return 0;
第28行  }
```

第五节　运算符重载

1、认识运算符

C++中预定义的运算符的操作对象只能是基本数据类型。但实际上，对于许多用户自定义类型（例如类），也需要类似的运算操作。这时就必须在C++中重新定义这些运算符，赋予已有运算符新的功能，使它能够用于特定类型执行特定的操作。运算符重载的实质是函数重载，它提供了C++的可扩展性，也是C++最吸引人的特性之一。

运算符重载是通过创建运算符函数实现的，运算符函数定义了重载的运算符将要进行的操作。运算符函数的定义与其他函数的定义类似，惟一的区别是运算符函数的函数名是由关键字operator和其后要重载的运算符构成。

和普通函数相比，重载运算符的函数参数不能有默认值，否则就改变了运算符的参数个数。重载的运算符只能是用户自定义类型，否则就不是重载而是改变了现有的C++标准数据类型的运算符的规则。户自定义类的运算符一般都必须重载后方可使用，但运算符“=”无需重载即可使用。

2、运算符重载位置

运算符重载可以通过成员函数的形式，也可是通过友元函数，非成员非友元的普通函数三种方式。例程2-12是运算符重载的三种形式。

例程2-12

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   class numUpDown{
第5行   public:
第6行       numUpDown(int Up=1,int Down=1){
第7行           this->upNum=Up;
第8行           this->downNum=Down;
第9行       };
第10行      numUpDown &operator+(int i){
第11行          this->upNum+=i*this->downNum;
第12行          return *this;
第13行      }
第14行      numUpDown &operator+=(int i){
第15行          this->upNum=this->upNum+this->downNum*i;
第16行          return *this;
第17行      }
第18行      friend numUpDown operator+(int i,numUpDown &rUD);
第19行      numUpDown &operator=(int i){
第20行          this->upNum=i;
第21行          this->downNum=i;
第22行          return *this;
第23行      }
第24行      void print(){
第25行          cout<<this->upNum<<"/"<<this->downNum<<endl;
第26行      }
第27行      int getUp(){
```

```
第28行            return this->upNum;
第29行        }
第30行        int getDown(){
第31行            return this->downNum;
第32行        }
第33行    private:
第34行        int upNum;
第35行        int downNum;
第36行    };
第37行    numUpDown operator+(numUpDown &left,numUpDown &right){
第38行        return numUpDown(left.getUp()*right.getDown()+right.getUp()*left.getDown(),left.getDown()*right.getDown(
    ));
第39行    }
第40行    numUpDown operator+(int i,numUpDown &rUD){
第41行        return numUpDown(i*rUD.downNum+rUD.upNum,rUD.downNum);
第42行    }
第43行    int main(){
第44行        numUpDown UD0(2,3);

第45行        numUpDown UD1=UD0+2;
第46行        numUpDown UD2=2+UD0;
第47行        numUpDown UD3=UD1+UD2;
第48行        system("pause");
第49行        return 0;
第50行    }
```

3、赋值运算符重载

与拷贝构造函数一样，如果类没有重载其赋值运算符，编译器会根据需要自动合成一个，即赋值运算符即便没有重载定义，也同样可以使用。赋值是二元运算，所以该操作符函数有两个形参，第一个形参对应左操作数，第二个形参对应右操作数。大多数操作符可以定义为函数成员或非函数成员，而赋值运算符则必须定义为函数成员，其this绑定到指向操作数的指针。鉴于此，赋值运算符重载函数接受单个形参，且该形参必须是与类相同类型的形参。

例程2-13

```
第1行    #include<iostream>
第2行    using namespace std;
第3行
第4行    class numUpDown{
第5行    public:
第6行        numUpDown(int Up=1,int Down=1){
第7行            std::cout<<"...Constructor..."<<std::endl;
第8行            this->upNum=Up;
第9行            this->downNum=Down;
第10行        }
第11行        numUpDown(numUpDown &UD){
第12行            std::cout<<"...Copy Constructor..."<<std::endl;
第13行            this->upNum=UD.upNum;
第14行            this->downNum=UD.downNum;
```

```
第15行      }
第16行      numUpDown& operator=(numUpDown &UD){
第17行          std::cout<<"...Operator=..."<<std::endl;
第18行          this->upNum=UD.upNum;
第19行          this->downNum=UD.downNum;

第20行          return *this;
第21行      }
第22行  private:
第23行      int upNum,downNum;
第24行  };
第25行  int main(){
第26行      numUpDown UD0(2,3);//调用构造函数
第27行      numUpDown UD1=UD0;//调用拷贝构造函数
第28行      UD1=UD0;//调用赋值运算符重载
第29行
第30行      return 0;
第31行  }
```

赋值运算符与拷贝构造函数类似。一旦需要显式赋值运算符重载，则右操作数对象的每个成员赋值给左操作数对象的对应成员，除数组、指针外，每个成员用所属类型的常规方式进行赋值。对于数组等，则需要每个数组成员赋值。

赋值运算符重载与拷贝构造函数经常一起使用。在实际应用中，常将其视为一个整体，如果需要其中一个，则几乎肯定需要另外一个。

4、下标运算符重载

下标运算符如果需要重载，必须重载为成员函数。下标运算符在操作系列数据时，直观友好，因此在序列数据中，常常重载下标运算符。如例程2-14所示。

例程2-14

```
第1行   #include<iostream>
第2行   using namespace std;
第3行   class Array{
第4行   public:
第5行       Array(int size=50){
第6行           this->size=size;
第7行           pointer=new int[size];
第8行       }
第9行       int & operator[](int i){
第10行          return *(pointer+i);
第11行      }
第12行  private:
第13行      int size;
第14行      int *pointer;
第15行  };
第16行  int main(){
第17行      Array myArr(500);
第18行
第19行
```

```
    for(int i=0;i<500;++i)myArr[i]=i;
第20行
第21行     for(int i=0;i<500;++i)cout<<myArr[i]<<endl;
第22行
第23行     system("pause");
第24行     return 0;
第25行 }
```

5、括号运算符重载

括号运算符如果需要重载，必须重载为成员函数。括号运算符重载后如同函数，称之为函数对象，又称仿函数，如例程2-15所示，第15行中LT(*p)中的LT不是函数名称，实际上是lessThan的圆括号运算符重载。采用这种模式，比函数更加灵活，其参数由结构中的数据成员提供，相关计算尤其括号运算符重载处理。括号运算符重载，是STL的重要组成部分，也是C++灵活性和技巧的表现。

例程2-15

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  struct lessThan{
第4行      int Arg1;//第一个参数
第5行      lessThan(int i){
第6行          this->Arg1=i;
第7行      };
第8行      bool operator()(int i){
第9行          return i<Arg1;
第10行     }
第11行 };
第12行 template<typename T>

第13行 void printIF(T *Head,T *Tail,lessThan LT){
第14行     for(T *p=Head;p!=Tail;){
第15行         if(LT(*p))std::cout<<*p<<std::endl;
第16行         ++p;
第17行     }
第18行     //注意：LT(*p)不是函数，而是lessThan的括号运算符重载
第19行 }
第20行 int main(){
第21行     int myArr[]={2,4,1,3,6,2,5,6,9,12,7};
第22行     int size=sizeof(myArr)/sizeof(myArr[0]);//数组长度
第23行     printIF(myArr,myArr+size,lessThan(5));
第24行     system("pause");
第25行     return 0;
第26行 }
```

第六节　引用与函数

返回指向函数调用前就已经存在的对象的引用是正确的。当不希望返回的对象被修改时，返回const引用是正确的。

例程2-16

```
第1行  #include<iostream>
第2行  using namespace std;
```

```
第3行
第4行  int TransA(int a){
第5行      a=a*10;
第6行      return a;
第7行  }
第8行  int TransB(int & a){//注意：参数a为引用方式
第9行      a=a*10;
第10行     return a;
第11行 }
第12行 int &TransC(int &a){
第13行     a=a*10;
第14行     return a;//写成return a*10错误
第15行 }
第16行 int &TransD(int a){
第17行     a=a*10;
第18行     return a;//a是临时变量，返回的是临时变量的地址
第19行 }
第20行 int main(){
第21行     int valA=9;
第22行     cout<<"TransA(valA)="<<TransA(valA);//TransA(valA)=90
第23行     cout<<"valA="<<valA<<endl;//valA=9，没有发生改变
第24行
第25行     cout<<"TransB(valA)="<<TransB(valA);//TransB(valA)=90
第26行     cout<<"valA="<<valA<<endl;//valA=90，TranB内的改变相当于直接改变valA
第27行     TransC(valA)=999;
第28行     //valA改变为999，TransC返回的valA的引用，相当于直接对valA赋值
第29行     cout<<valA<<endl;
第30行
第31行     TransD(valA)=111;
第32行     cout<<valA<<endl;//valA的值没有被改变，仍然是999;
第33行
第34行     system("pause");
第35行     return 1;
第36行 }
```

例程2-17

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class numUpDown{
第4行  public:
第5行      numUpDown(int Up=1,int Down=1){
第6行          this->upNum=Up;
第7行          this->downNum=Down;
第8行      }
第9行      numUpDown(numUpDown &UD){
```

```
第10行            std::cout<<"In COPY!!!"<<std::endl;
第11行            this->upNum=UD.upNum;
第12行            this->downNum=UD.downNum;
第13行        }
第14行        int *getAddrUp(){
第15行            return &this->upNum;
第16行        }
第17行        int &getUp(){
第18行            return this->upNum;
第19行        }
第20行        int getDown(){
第21行            return this->downNum;
第22行        }
第23行        void print(){
第24行            std::cout<<"("<<this->upNum<<"/"<<this->downNum<<")"<<std::endl;
第25行        }
第26行    private:
第27行        int upNum,downNum;
第28行    };
第29行    void printA(numUpDown UD){
第30行        std::cout<<UD.getUp()<<"/"<<UD.getDown()<<std::endl;
第31行    }
第32行    void printB(numUpDown &UD){
第33行        std::cout<<UD.getUp()<<"/"<<UD.getDown()<<std::endl;
第34行    }
第35行    int main(){
第36行        numUpDown UD0(2,3);
第37行        UD0.getUp()=10;//直接改变了private中的upNum;
第38行        //UD0.getDown()=11;//注意：本语句错误，由于没有返回引用。
第39行        UD0.print();//输出(10/3);
第40行        std::cout<<UD0.getUp()<<std::endl;
第41行        printA(UD0);//输出In COPY!!!后输出10/3
第42行        printB(UD0);//没有输出In COPY!!!后输出10/3
第43行        *UD0.getAddrUp()=7;
第44行        UD0.print();
第45行        system("pause");
第46行        return 1;
第47行    }
```

第七节　类中应用const

const是英文单词constant的简写，其含义是“常数、常量、不变的事物、永恒值”，在C++程序中，常用于不允许变化的场所，如例程2-18所示，其第15、22行的函数名称int getUp()const和int getDown()const后均出现const关键字，其功能是在函数作用范围内不能修改类的数据成员，实现对数据成员的保护，防止函数代码修改数据成员的值。

例程2-18

```
#include<iostream>
using namespace std;
class numUpDown{
public:
    numUpDown(int Up=1,int Down=1){
        this->upNum=Up;
        this->downNum=Down;
    }
    void setUp(int Up){
        this->upNum=Up;
    }
    //const的使用表明不允许类的数据成员被修改
    //int不能修改为int &，const的修饰符的使用将const化数据成员
    //存在将const int转换为int &的错误
    int getUp()const{
        return this->upNum;
    }
    int getUp(){
        return this->upNum;
    }
    //没有const，可以修改this->downNum的值；
    int getDown()const{
        std::cout<<"...const!"<<std::endl;
        return this->downNum;
    }
    int &getDown(){
        std::cout<<"...NO const!"<<std::endl;
        return this->downNum;
    }
private:
    int upNum,downNum;
};
//注:函数名不能使用const修饰符，非成员函数不能使用类型修饰符
//参数使用const限定形参，表明函数内部不能修改该参数的值
void printUD(const numUpDown &UD){
    //UD.setUp(10);//注：错误！对象包含与成员函数不兼容的类型限定符
    std::cout<<UD.getUp()<<"/"<<UD.getDown()<<std::endl;
}
int main(){
    numUpDown UD0(2,3);
    std::cout<<UD0.getDown()<<std::endl;//会输出...NO const!
    printUD(UD0);//会输出...const!
    UD0.getDown()=5;//会输出...NO const!
    printUD(UD0);//会输出...const!
```

```
第45行      system("pause");
第46行      return 0;
第47行  }
```

注意观察第15-17行与第18-20行以及第22-25行与第26-29行，会发现getUp()函数除const外完全相同，其原因是const也是实现函数重载的要素。即如果函数参数完全相同，但有const区别，也是不同的函数。在观察main()函数能发现，程序能自动选择正确的函数形式。

当const出现在函数的参数前时，将保护参数不能被修改，如例程2-18第27行void printUD(const numUpDown &UD)所示。在该函数中，const的应用将使形参UD在printUD函数中不能被修改，是UD值的保护。在实际编程过程中，如果输入的参数不希望被修改，加上const是良好的工程习惯。

当const与指针变量一起使用时，情况将变得略微复杂，如例程2-19所示，const可以出现符号*之前，也可以在其后。前后位置不同，其含义也不同。

例程2-19

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  int main(){
第5行      int numZ=100;
第6行      int numA=200;
第7行      int numB=300;
第8行      int numC=400;
第9行
第10行     int *pZ=&numZ;//获得numZ的地址
第11行     int *const pA=&numA;//获得numA的地址
第12行     const int *pB=&numB;//获得numB的地址
第13行     const int *const pC=&numC;//获得numC的地址
第14行
第15行     pZ=&numA;//注意：允许!pZ改指向numA
第16行     //pA=&numB;//注意：错误!pA的地址值为常量，不能修改，即不能调整为numB的地址;
第17行     pB=&numA;//注意：允许!pB的地址值不为常量，可以指向其他地址;
第18行     //pC=&numA;//注意：错误!pC的地址值为常量，不能修改，即不能调整为numA的地址;
第19行     cout<<"*pZ="<<*pZ<<" numZ="<<numZ<<endl;
第20行     cout<<"*pA="<<*pA<<" numA="<<numA<<endl;
第21行     cout<<"*pB="<<*pB<<" numB="<<numB<<endl;
第22行     cout<<"*pC="<<*pC<<" numC="<<numC<<endl<<endl;
第23行
第24行     *pZ=111;//注意：常规应用，可以修改内容，可以改变地址，在前面pZ已指向numA
第25行     *pA=222;//注意：不可改变地址，但可以改变内容
第26行     //*pB=333;//注意：可以改变地址，不可改变内容，在前面pB已指向numB
第27行     //*pC=444;//注意：不可改变地址，不可改变内容
第28行     cout<<"*pZ="<<*pZ<<" numZ="<<numZ<<endl;
第29行     cout<<"*pA="<<*pA<<" numA="<<numA<<endl;
第30行     cout<<"*pB="<<*pB<<" numB="<<numB<<endl;
第31行     cout<<"*pC="<<*pC<<" numC="<<numC<<endl<<endl;
第32行
```

```
第33行        //num系列变量值变化不受影响
第34行        numZ=199;
第35行        numA=299;
第36行        numB=399;
第37行        numC=499;
第38行        cout<<"*pZ="<<*pZ<<" numZ="<<numZ<<endl;
第39行        cout<<"*pA="<<*pA<<" numA="<<numA<<endl;
第40行        cout<<"*pB="<<*pB<<" numB="<<numB<<endl;
第41行        cout<<"*pC="<<*pC<<" numC="<<numC<<endl<<endl;
第42行
第43行        return 0;
第44行    }
```

当const用于修饰函数的返回值时，如const int f1()相当于const int value=f1()，即函数的返回值为常量，不能被修改。当const用于修饰的函数返回值带有指针时，形如const int *f1()、int *const f2()或const int *const f3()对应相当于const int *p1=f1()、int *const p2=f2()或const int *const p3=f3()，其含义与变量相似，在例程2-19中，其地址来源于变量，用于修饰函数返回值时，则地址来源函数返回值。

当const用于修饰函数的参数时，其含义与例程2-19相似，都是变量的地址。