1. 定义类与对象

class	缺省说明时，其成员被认为是私有的
struct	若不特别指出，其所有成员都是公有的
union	其所有成员都是公有的，且不能更改

类由成员构成：

1. 数据成员————描述对象的属性
2. 成员函数————描述对象的方法

注：

1. 允许已定义类出现在类的说明中

```
class link
{
   link *next;
   ……
};


class X
{
     ……
};
class Y
{
    X datamenber;  //声明一个类类型的数据成员
    ……
}
```

2. 类可以无名，用于直接声明对象
3. 类是一个程序包，可以只有数据成员或只有成员函数，或者为空

2.访问对象成员

2.1 指针 动态存储

```
Date *pd =new (Date);
pd->Setdate(2001,10,1);
pd->print();
delete pd;
```

```
#include<iostream>
using namespace std;
class Tclass
{
public:
    int x, y;
    void out()
    {
        cout << x << " " << y << endl;
    };
};
int add(Tclass* ptf)
{
    return (ptf->x + ptf->y);
}
int main()
{
    Tclass test, * pt = new(Tclass);
    pt->x = 100; pt->y = 200;
    pt->out();
    cout << "x+y=" << add(pt) << endl;
    test.x = 150; test.y = 450;
    test.out();
    cout << "x+y=" << add(&test);
}
```

2.2 this指针

成员函数拥有this指针,隐含指针，不能显示说明，但可以在成员函数中显示使用，this指针是一个常指针

```
Person & addage(Person p)
{
    this->age+=p.age;
    return *this;
}

int main()
{
    ……
    p2.addage(p1).addage(p1);
    //链式思维
}
```

3.构造函数与析构函数

3.1 构造函数的初始化

```
Date :: Date(int y,int m,int d)
{
    year=y; month=m; day=d;
}
```

```
Date::Date(int y,int m,int d):year(y),month(m),day(d) {}
```

有时候只能使用第二种初始化，如下 初始式 （参数初始化）的初始化：

1. 类成员的初始化
2. 派生类的初始化
3. 常成员的初始化

```
#include<iostream>
using namespace std;
class A
{
public:
    A(int x) :a(x) {}
    int a;
};
class B
{
public:
    B(int x,int y):aa(x),b(y){}
    void out()
    {
        cout << "aa=" << aa.a << endl << "b=" << b << endl;
    }
private:
    int b;
    A aa;
};
int main()
{
    B objB(3, 5);
    objB.out();
    return 0;
}
```

3.2重载构造函数

```
class X
{
    public:
```

```
    X();
    x(int);
    X(int ,char);
    X(double ,char);
}

void f()
{
    X a;
    X b(1);
    X c(1,'c');
    X d(2.3,'d');
}
```

3.3深拷贝和浅拷贝

拷贝构造函数：

类名：：类名（const 类名 & 应用名 ）；

深拷贝

```
#include<iostream>
#include<cstring>
using namespace std;
class Name
{
public:
    Name(const char* pn);
    Name(const Name& oBj);
    ~Name();
    void setName(const char*);
    void showName();
protected:
    char* pName;
    int size;
};
Name::Name(const char* pn)            //定义拷贝构造函数
{
    cout << "Constructing" << pn << endl;
    pName = new char[strlen(pn) + 1];
    if (pName != 0)strcpy_s(pName, strlen(pn) + 1, pn);
    size = strlen(pn);
}
Name::Name(const Name& obj)
{
    cout << "Copying " << obj.pName << "into its own block\n";
    pName = new char[strlen(obj.pName) + 1];
    if (pName != 0)strcpy_s(pName, strlen(obj.pName) + 1, obj.pName);
    size = obj.size;
}
Name::~Name()
```

```
{
    cout << "Destroying " << pName << endl;
    pName[0] = '\0';
    delete[]pName;
    pName = NULL;
    size = 0;
}
void Name::setName(const char* pn)
{
    delete[]pName;
    pName = new char[strlen(pn) + 1];
    if(pName!=0)strcpy_s(pName, strlen(pn) + 1, pn);
    size = strlen(pn);
}
void Name::showName()
{
    cout << pName << endl;
}
int main()
{
    Name Obj1("noname");
    Name Obj2 = Obj1;
    Obj1.showName();
    Obj2.showName();
    Obj1.setName("sushi");
    Obj2.setName("dufu");
    Obj1.showName();
    Obj2.showName();
}
```

4. 常成员

4.1 常数据成员

const Person p;

用参数初始式对常数据成员赋值

```
class Mclass
{
public:
    int k;
    const int M;
    Mclass () M(5){}        //用参数初始式对常数据成员赋值
};

//不能在成员函数和类外修改常数据成员
```

用带参数构造函数完成数据成员初始化

```
Student::Student(int y,int m,int d,int num,char *pName):code(num)
{
    strcpy_s(name,pName);
    name[sizeof(pName)-1]='\0';
    birthday.year=y;
    birthday.month=m;
    birthday.day=d;
}
```

4.2常对象

常对象只能调用常函数

```
class Tclass
{
    public:
    int x,y;
    Tclass(int a,int b)
    {
        x=a,y=b;
    }
};

int main()
{
    const Tclass t1(1,2);  //常对象,不可修改其数据成员
}
```

4.3常函数

1. 成员函数后加const后称该函数为常函数
2. 常函数不可以修改成员属性
3. 成员属性声明时加关键字mutable，在常函数中依然可以修改

```
#include<iostream>
using namespace std;
class student
{
    int a;
    int  mutable b;
public:
    student (int x,int y): a(x),b(y){}
    void func1()
    {
        a++;
        b++;
        cout << a << " " << b << endl;
    }
```

```
    void func2()const
    {
        //a++;错误，常函数不可以修改成员属性
        b++;
        cout << a << " " << b << endl;
    }

};
int main()
{
    student s(1, 2);
    s.func1();
    s.func2();

}
```

5.静态成员

5.1 静态数据成员

1. 存储性质：全局变量 （同类对象的共享）
2. 作用域是类，在类中的public,protected,private起约束访问作用
3. 声明static数据成员static int num 不会建立存储空间，需要在类声明外由static数据成员的说明语句，初始化

（*类外定义并初始化*） int Counter::num=0;

4. 访问 可以通过类名或者对象访问 Counter::Num; 或 c.Num;

```
#include<iostream>
using namespace std;
class student
{
public:
    static int snum;
    int num;
    student(int b) :num(b){}
};
int student:: snum = 0;         //类外定义并初始化
int main()
{
    student s(2);
    for (int i = 0; i < 5; i++)
    {
        student::snum += i;
        cout << student::snum << '\t';   //通过类名访问
    }
    cout << endl;
    cout << "s.snum=" << s.snum << endl;      //通过对象访问
```

```
    cout << "s.num=" << s.num << endl;          //通过对象访问
    return 0;
}
```

5.2静态成员函数

没有this指针，静态成员函数只能访问类的静态数据成员

类外访问，可以用"类名：："作为限定词，或通过对象调用

```
class X
{
public:
    static void stafun(int i,X*ptr);
    int stadat;
}
void X::stafun(int i,X *ptr)
{
    ptr->stadat=i;          //通过参数访问数据成员
}
int main()
{
    X obj;
    X::stafun();       //通过类名访问
    obj.stafun();     //通过对象访问
}
```

购买货物

```
#include<iostream>
using namespace std;
class goods
{
private:
    int weight;
    static int total_weight;
public:
    goods(int w)
    {
        weight = w;
        total_weight += w;
    }
    goods* next;
    ~goods()
    {
        total_weight -= weight;
    }
    int Weight()
    {
```

```
        return weight;
    }
    static int Totalweight()
    {
        return total_weight;
    }
};
int goods::total_weight = 0;
void purchase(goods* & f,goods*&r,int w)
{
    goods* t=new goods(w);
    t->next = NULL;
    if (f == NULL)f =r= t;
    else
    {
        r->next = t;
        r = r->next;
    }
}
void sale(goods* &f, goods* &r)
{
    if (f == NULL)
    { cout << "No any goods now.\n";
    return; }
    goods* p = f;
    f = f -> next;
    delete p;
    p = NULL;
    cout << "sold\n";

}
int main()
{
    goods* front = NULL, * rear = NULL;
    int w, choice;
    do
    {
        cout << "Please choose\n";
        cout << "Key in 1 is purchase,key in 2 is sale,key in 0 is over.\n";
        cin >> choice;
        switch (choice)
        {
        case 1:
        {
            cout << "Input weight:";
            cin >> w;
            purchase(front, rear, w);
            break;
        }
        case 2:
        {
            sale(front, rear);
            break;
        }
```

```
        case 0:
            break;
        }
        cout << "Now total weight is " << goods::Totalweight() << endl;
    } while (choice);
}
```

6.友元

1. 友元可以访问类的所有成员，包括私有成员，友元可以是一个普通函数，成员函数或一个类，友元关系是非对称的、非传递的。

2. 友元函数

    1. 友元函数不受private,public,protected的影响，友元函数不是类的成员，仅作友元声明

```
class A
{
 private:
     int i;
    void MemberFun(int);
    friend void FriendFun(A*ptr,int);
}
void Friendfun(A*ptr,int a)
{
      ptr->i=a;           //友元函数通过参数访问类的私有成员
}
void MemberFun(int x)
{
      this->i=x;        //成员函数通过this指针访问类的私有成员
}
```

    2. 友元函数必须在参数表中显示指明要访问的对象

```
A Aobj;
FriendFun(&obj,5);
Member(5);
```

7.1运算符重载规则

1. 不能重载的运算符：

```
.    .*    ::   ?:  sizeof
```

C++中不能用友元函数重载的运算符有：

```
= () [] ->
```

不能改变运算符的优先级、结合性、所需要操作的个数

模仿数学运算符 矩阵、复数、向量

默认重载：

1. 赋值运算符 = 默认重载为对象数据成员的复制
2. 地址运算符 & 默认重载为返回任何类对象的地址

7.2用成员函数或友元函数重载运算符

1.

区别	**成员函数**	**友元函数**
关键	有this指针	无this指针
一元运算符	Object op()	operator op(Object)
二元运算符	ObjectL op(ObjectR)	operator op(ObjectL,ObjectR)
	一元运算符的操作数、二元运算符的操作数是该类的一个对象	运算符左右操作数类型不同，二元运算交换性?
运算符操作需要修改类对象状态	用成员函数， 如= 、*= 、++	若用友元函数，使用*引用*参数修改对象
运算符操作数有隐式转换（尤其是第一个操作数）	无	用友元函数或普通函数重载
注意	使用成员函数重载 = () [] ->	C++中不能用友元函数重载的运算符有：= () [] -> ； 但是<< 与 >>只能用友元函数

2. 用成员函数重载运算符

```
class TriCoor
{
    public:
       TriCoor (int max=0,int my=0,int mz=0);
       TriCoor operator+(TriCoor t);
       TriCoor & operator=(TriCoor t);
    private:
       int x,y,z;
};
```

```
TriCoor TriCoor::operator+(TriCoor t)
{
   TriCoor temp;
   temp.x=this->x+t.x;
   temp.y=this->y+t.y;
   temp.z=this->z+t.z;
   return temp;               // a+b+c  被解释为  (a+b)+c
}
TriCoor & TriCoor::operator=(TriCoor t)
{
    this->x=t.x;
    this->y=t.y;
    this->z=t.z;
    return *this;       //a=b=c 被解释为 a=(b=c);
             // 返回类型为类对象，同时是 引用& ,可节省匿名空间
            //连续运算，使得重载运算符满足原来的语义
}
```

3.用友元函数重载运算符

```
class Complex
{
  public:
    Complex(double r=0,double i=0);
    Complex(int a)
    {
        Real=a;
        Image=0;
    }
    friend Complex operator+(const Complex &c1,const Complex &c2);

  private:
     double Real,Image;
};
Complex operator+(const Complex &c1,const Complex &c2)
{
    double r=c1.Real+c2.Real;
    double i=c1.Image+c2.Image;
    return Complex(r,i);    //显式调用构造函数
}

int main()
{
    Complex c1(2.4,3.4),c;
    c=25+c1;     //将25int类型转变为类类型
                 //同时重载函数要const &才能使用常数
}
```

运算符操作需要修改类对象状态 用成员函数， 如= 、*= 、 ++。 若用友元函数，使用引用参数&修改对象

```
TriCoor operator++(TriCoor & op1)
{
    op1.x++;
}

TriCoor ob(1);
ob++;
```

7.3几个经典运算符的重载

1. 重载++与--运算符

```
class Increase
{
   public:
      Increase();
      ……
      //成员函数
      Increase operator++();                    //++a
      Increase operator++(int);                 //a++
      //友元函数
      friend Increase operator++(Increase &);           //++a
      friend Inrease  operator++(Increase & ,int);      //a++

};
//成员函数 ++a
Increase Increase:: operator++()
{
    value++;
    return *this;
}

//成员函数a++
Increase Increase ::operator++(int)
{
    Increase temp;
    temp.value=value++;
    return temp;
}

//友元函数++a
Increase operator++(Increase &a)
{
    a.value++;
    return a;
}

//友元函数a++
Increase operator ++ (Increase &a,int)    //int是伪参数
```

```
{
    Increase temp(a);
    a.value++;
    return temp;
}
```

2. 重载赋值运算符=

用于对象数据的复制，只能用于成员函数重载，而且不能被继承

```
Name::Name(const Name & obj)         //定义拷贝构造函数
{
   cout<<"Copying "<<Obj.pname<<"into its owm block\n";
   size=obj.size;
   pname=new char[size+1];
   if(pname!=0)strcpy_s(pname,size+1,obj.name);
}
Name Name:: operator=(Name obj)        //重载赋值运算符
{
    delete []pname;      //! ! !
    size=obj.size;
    pname=new char[size+1];        //要先申请存储空间
    if(pname!=0)strcpy_s(pname,size+1,obj.name);
    return *this;
}


int main()
{
    Name obj1("ZhangSan");
    Name obj2=obj1;           //拷贝构造函数用于对象的初始化
    Name obj3("Noname");
    obj3=obj2=obj1;          //用于程序运行时修改对象的程序
}
```

3.重载[]和()运算符

```
int & Vector :: operator[](int i)const
{
    if(i>=0 && i<len)return v[i];
    cout<<"The subscript "<<i<<"is outside.\n";
    exit(0);
}

//其中int & operator[](int i)const;
// 当重载运算符函数调用需要作为左值时，应该返回引用
//A[i]=i+1;

int Vector::operator()()const
```

```
{
    return len;
}
```

4.重载流插入<<和流提取>>运算符

cout是C++输出流ostream的预定义对象，用于连接显示器；cin是输入流istream的预定义对象，用于连接键盘，不是Vector类对象。

使用&是为了链式调用

```
class Vector
{
   friend ostream & operator<<(ostream output ,conse Vector &);
    friend istream & operator>>(istream input ,conse Vector &);
};


//输出向量
ostream & operator<<(ostream & output,const Vector & A)
{
    for(int i=0;i<A.len;i++)
        output<<A.v[i]<<" ";
    return output;       //output 其实是cout的别名，直接写cout也行
}


//输入向量
istream & operator>>(istream &input,const Vector &A)
{
    for(int i=0;i<A.len;i++)
       input>>A.v[i];
    return input;
}                        //input 其实是cin的别名，直接写cin也行
```

7.4类类型的转换

可以使用构造函数和类型转换函数，有隐式调用和显式调用

1.使用构造函数

具有一个非默认参数的构造函数实现从参数类型到该类类型的转换。

```
ClassX :: ClassX(arg,agr1=e1,……);
```

```
Complex (int a)
{
    Real=a;Image=0;
}

类型转换 :
void funx(Complex);        //函数原型
Complex x=Complex (3);     //显式把整型3转换为Complex对象
Complex y=5;               //对5进行类型转换，作为y的初值
x=12;                      //赋值的类型转换
funx(27);                  //参数传递的类型转换
```

2.使用类型转换函数

具有一个非默认参数的构造函数能够把某种类型对象转换为指定类对象，但不能将一个类对象转换为基本数据类型值。

所以引入类型转换函数 （特殊的成员函数）

```
ClassX ::operator Type()
{
    //……
    return Type_Value;
}
```

这个函数功能就是把ClassX类型对象转换为Type类型的对象。类型转换函数没有参数，没有返回类型，但必须有一个返回Type类型值的语句。

类型转换函数只能定义为一个类的成员函数，不能定义为类的友元函数。

```
class Calculator
{
    public:
       operator int ()
       {
            int a;
            a=value;
            return a;
       }
       private:
          int value;
}


int main()
{
    Caculator Counter;
    Counter++;
```

```
    cout<<int(Counter)<<endl;
}
```

1.基类与派生类

访问控制		
公有继承	基类的公有成员->派生类的公有成员	基类的保护成员->派生类的保护成员；
私有继承	基类的公有和保护成员->派生类的私有成员	
保护继承	基类的公有和保护成员->派生类的保护成员	
	基类的私有成员，在派生类中有存储空间，但不能直接使用，得通过成员函数访问	
	protected成员在类体系中 派生类可见，类外不可见	

2.重名成员

在派生类中访问重名成员时，屏蔽基类的同名成员。如果要在派生类中使用基类的同名成员，通过（类名：：成员）访问。

3.派生类中访问静态成员

1. 基类定义的静态成员，将被所有派生类共享
2. 根据静态成员自身的访问特性和派生类的继承方式，在类层次体系中具有不同的访问性质
3. 在派生类中可以通过类名显示访问 或者对象访问

```
类名：：成员
对象.成员
```

4.基类的初始化

派生类对象的初始化（初始式）

派生类构造函数声明为

```
派生类构造函数（变元表）：基类（变元表），对象成员（变元表）.....对象成员（变元表）；
```

初始式 （参数初始化）的初始化：

1. 类类型成员的初始化

2. 派生类的初始化
3. 常成员的初始化

构造函数执行顺序：基类->对象成员->派生类

```
#include<iostream>
using namespace std;
class parent_class
{
    int data1, data2;
public:
    parent_class(int p1, int p2) { data1 = p1; data2 = p2; }
    int inc1() { return ++data1; }
    int inc2() { return ++data2; }
    void display()
    {
        cout << "data1=" << data1 << " data2=" << data2 << endl;
    }
};
class derived_class :private parent_class
{
    int data3;
    parent_class data4;
public:
    derived_class(int p1, int p2, int p3, int p4, int p5) :
        parent_class(p1, p2), data4(p4, p5)
    {
        data3 = p3;
    }
    int  inc1() { return parent_class::inc1(); }
    int inc3() { return ++data3; }
    void display()
    {
        parent_class::display();
        data4.display();
        cout << "data3=" << data3 << endl;
    }

};
int main()
{
    derived_class d1(17, 18, 3, 5, 6);
    d1.inc1();
    d1.display();
    return 0;
}
```

5.多继承

多继承中，多个基类构造函数执行顺序取决于 *定义派生类时指定的各个继承基类的顺序

```
class Derived:public Base1,public Base2
{
    ……
}
```

6.虚继承 virtual

使公共基类在派生类中只产生一个对象，必须对这个基类声明为虚继承，使得这个基类称为虚基类

```
class B
{public:int b};
class B1:virtual public B
{
    private:int b1;
};
class B2:virtual public B
{
    private:int b2;
};
class C:public B1,public B2
{
    private int  d;
}


C cc;
cc.b;   //right
```

虚函数与多态性

1.类指针的关系

1.用基类指针访问派生类对象

在main函数中，基类指针A_p获取派生类对象B_obj的地址之后，可以调用基类的成员函数对派生类对象进行访问，但是，不能直接用A_p访问派生类自己定义的成员函数。 需要通过强制类型转换来演示对派生类自己定义成员函数的调用

```
( (B_class*) A_p)->show_phone();
```

2.用派生类指针访问基类对象 （强制类型转换）

```
class Date{};
class DateTime:public Date{};
```

```
DateTime dt(2003,1,1,12,30,0);
DateTime *pdt=&dt;
((Date)dt).print();              //对象类型强制转换，把派生类对象强制转换为基类对象
dt.print();
((Date*)pdt)->print();    //对象指针类型强制转换，把派生类对象指针强制转换为基类对象指
针
pdt->print();
```

2.虚函数和动态联编

以virtual关键字开头的成员函数称为虚函数

虚函数不能是构造函数

```
#include<iostream>
using namespace std;
class Base
{
protected:
    char x;
public:
    Base(char xx) { x = xx; }
    virtual void who()         //说明虚函数
    {
        cout << "Base:" << x << endl;
    }
};
class First_d :public Base
{
protected :
    char y;
public:
    First_d(char xx, char yy) :
        Base(xx) {
        y = yy;
    }
    void who()      //默认说明虚函数
    {
        cout << "First_d" << y << endl;
    }
};
class Second_d :public First_d
{
protected:
    char z;
public:
    Second_d(char xx, char yy, char zz) :
        First_d(xx, yy) {
        z = zz;
    }
```

```
    void who()          //默认说明虚函数
    {
        cout << "Second_d" << z << endl;
    }
};
int main()
{
    Base b('A');;
    Base* p = &b;
    p->who();
    First_d f('a', 'b');
    p = &f;
    p->who();
    Second_d s('b', 'c', 'd');
    p = &s;
    p->who();
    return 0;
}
```

虚析构函数

3.纯虚函数和抽象类

纯虚函数是在基类中说明但没有定义的虚函数，要求所有的派生类都必须提供自己的版本。

```
virtual 类型 函数名（参数表）=0;
```

一个具有纯虚函数的基类称为抽象类。

1. 抽象类只能用作其他类的基类
2. 抽象类不能建立对象
3. 但可以声明抽象类的指针（可以指向其派生类对象）
4. 抽象类不能作为函数返回类型，参数类型或显示类型转换

```
#include<iostream>
using namespace std;
class Figure
{
protected:
    double x, y;
public:
    void set_dim(double i, double j = 0)     //默认值为圆的半径 y=0
    {
        x = i; y = j;
    }
    virtual void show_area()const = 0;  //定义纯虚函数
};
class Triangle :public Figure
{
```

```
public:
    void show_area()const
    {
        cout << "Triangle with high" << x << " and base " << y;
        cout << "has an area of " << x * 0.5 * y << endl;
    }
};
class Square :public Figure
{
public:
    void show_area()const
    {
        cout << "Square with dimension " << x << "*" << y;
        cout << "has an area of " << x * y << endl;
    }
};
class Circle :public Figure
{
public:
    void show_area()const
    {
        cout << "Circle with radius " << x;
        cout << "has an area of" << 3.14 * x * x << endl;
    }
};
int main()
{
    Figure* p;
    Triangle t;
    Square s;
    Circle c;
    p = &t;
    p->set_dim(10.0, 5.0);
    p->show_area();
    p = &s;
    p->set_dim(8.0, 6.0);
    p->show_area();
    p = &c;
    p->set_dim(9.0);
    p->show_area();
    return 0;
}
```

5.异质链表

动态异质链表，需要在抽象类定义中增加一个公有数据成员，指针成员next

```
class Employee
{
public:
    Employee(const long,const char *);
```

```
    Employee *next;
protected:
    long number;
    char name[20];
}

void AddFront(Employee *&h,Employee *&t)
{                 //指针引用
    t->next=h;
    h=t;
}

void test3()
{
    Employee *empHead=NUll,*ptr;
    ptr=new Manager(10135,"Cheng ming",1200);                 //建立第一个结点
    AddFront(empHead,ptr);                                    //插入表头
    ptr=new HourlyWorker(30712,"zhao xiaoming",2,8*20);       //建立第二个结点
    AddFront(empHead,ptr);                                   //插入表头
    ptr=new PieceWorker(20382,"Xiu luwei",0.5,2850);          //建立第三个结点
    AddFront(empHead,ptr);                                    //插入表头
    ptr=empHead;
    while(ptr)
    {                                                              //遍历链表
        ptr->print();
        ptr=ptr->next;
    }
    ptr=empHead;
    while(ptr)
    {
        cout<<ptr->getName()<<" "<<ptr->earnings()<<endl;
        ptr=ptr->next;
    }
}
```

模板说明:

```
template <typename T1,typename T2, ……>
```

1.函数模板

```
#include<iostream>
using namespace std;
template <typename T>          //模板说明
T Max(const T a, const T b)     //函数定义（函数参数不仅是类属参数，也可以是一般类型参数)
{
    return (a > b ? a:b);            //将模板实例化，编译器生成模板函数
}
```

```
int main()
{
    int a, b;
    cout << "Input two integer.\n";
    cin >> a >> b;
    cout <<"max(" << a << "," << b << ")=" << Max(a, b) << endl;
    double c, d;
    cout << "Input two double .\n";
    cin >> c >> d;
    cout << "max(" << c << "," << d << ")=" << Max(c, d) << endl;
    cout << "Input two charater.\n";
    char e, f;
    cin >> e >> f;
    cout << "max(" << e << "," << f << ")=" << Max(e, f) << endl;
    return 0;
}
```

```
template <typename T>
void sortBubble(T* arr, int size)
{
    int i, work;
    T temp;
    for (int pass = 0; pass <= size - 1; pass++)
    {
        work = 1;
        for (int j = 0; j <= size - pass - 1; j++)
        {
            if (arr[j] < arr[j + 1])
            {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                work = 0;
            }
        }
        if (work)break;
    }
}
```

重载函数模板

2.类模板

类模板由模板说明和类说明构成

```
template<typename T>
class Array
{                                          //定义类模板
```

```
public:
    Array(int s);
    virtual ~Array();
    virtual const T& Entry(int index)const;
    virtual void Enter(int index, const T& value);
protected:
    int size;
    T* element;
};
template<typename T>Array<T>::Array(int s)
{
    if (s > 1)size = s;
    else size = 1;
    element = new T[size];
}
template<typename T>Array<T>::~Array()
{
    delete[]element;
}

int main()
{
    Array<int>IntAry(5);     //先实例化，建立模板类对象

}
```

3.类模板作为函数参数

函数是函数模板

```
template<typename T>
void Tfun(const Array<T>&x,int index)
{
    cout<<x.Enter(index)<<endl;
}

//要先类模板实例化
Array <double >DouAry(4);
Tfun(DouAry,3);
```

4.在类层次中的类模板

```
template<typename T>
class BoundArray:public Array<T>
{
public:
    BoundArray(int low=0,int height=1);
    //……
private:
```

```
    ……
}
```

从类模板派生模板类

```
template<typename T>
class A:
{
public:
    A T(x){t=x;}
    void out(){cout<<t<<endl;}
protected:
    T t;
}
class B:public A<int>           //实例化基类的类属参数，派生模板类
{
public:
    B(int a,double x):A<int>(a);
    {
        y=x;
    }
    void out()
    {
        A<int>::out();
        cout<<y<<endl;
    }
private:
    double y;
}
```

模板类的友元类

重载运算符的友元函数

```
template <typename T>
class Complex
{
    public:
     template <typename T>
     friend Complex <T>operator +(const Complex<T>c1,const Complex<T>c2);
}
```

5.类模板与静态成员

```
#include<iostream>
#include<string>
```

```
using namespace std;
template<class Nametype,class Agetype>
class Person
{
public:
    Person(Nametype name, Agetype age)
    {
        m_name = name;
        m_age = age;
        }
    void showperson()
    {
        cout << "Name:" << m_name << " age:" << m_age << endl;
    }
private:
    Nametype m_name;
    Agetype m_age;
};
int main()
{
    Person<string,int>p1("lucy", 19);
    p1.showperson();
    return 0;
}
```

5.文件操作

通过文件可以将数据永久化

头文件< fstream >

文件类型两种：

1. 文本文件：以ASCII码形式存储在计算机
2. 二进制文件 文本的二进制形式存储在计算机中，用户一般不能直接读懂它们

操作文件的三大类

1. ofstream：写操作
2. ifstream：读操作
3. fstream：读写操作

5.1写文件

1. 包括头文件

```
#include<fstream>
```

2. 创建流对象

```
ofstream ofs;
```

3. 打开文件

```
ofs.open("文件路径", 打开方式);
```

4. 写数据

```
ofs<<" ";
```

5. 关闭文件

```
ofs.close();
```

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置 文件末尾
ios::app	追加方式写文件
ios::trunc	如果文件存在先删除，再创建
ios::binary	二进制方式

注意：文件打开方式可以配合使用，利用 | 操作符

例如：用二进制写文件

```
ios::binary | ios::out
```

5.2读文件

1. 头文件

```
#include<fstream>
```

2. 创建对象

```
    ifstream ifs;
```

3. 打开文件并判断文件是否存在

```
    ifs.open("test.txt", ios::in);
    if (!ifs.is_open())
    {
        cout << "Failed.\n";
        return;
    }
```

4. 读数据

```
    char buf[1024] = { 0 };
    while (ifs >> buf)
    {
        cout << buf << endl;
    }
```

```
char buf[1024]={0};
while(ifs.getline(buf,sizeof(buf)))
{
    cout<<buf<<endl;
}
```

5. 关闭文件

```
    ifs.close();
```

5.3二进制写文件

```
class Person
{
public:
    char name[64];
    int len;
};
```

```
void test1()
{
    ofstream ofs("person.text", ios::out | ios::binary);          //也可以用这种方式构造
    //ofs.open("person.text", ios::out | ios::binary);
    Person p={"lucy",18};
    ofs.write((const char*)&p, sizeof(Person));
    ofs.close();
}
```

5.4二进制读文件

```
ifstream ifs;
    ifs.open("person.text", ios::in | ios::binary);
    if (!ifs.is_open())
    {
        cout << "Failed\n";
    }
    Person p;
    ifs.read((char*)&p, sizeof(Person));
    cout << "Name:" << p.m_name << "  Age:" << p.m_age << endl;
    ifs.close();
```

1.流类库

streambuf和ios类

ios类派生

1. istream (ifstream文件输入流类，istrstream字符串输入流类，istream_withassign重载赋值运算符的输入流类)
2. ostream(ofstream文件输出流类,ostrstream串输出流类,ostream_withassign重载运算符输出流类)

ostream有标准输出流cout\cerr(标准错误输出流) 、 clog（标准错误输出流，连接打印机）

2.输入流操作

函数	功能	示例
read	无格式输入指定字节数	istream read(char *pch,int ncount)
get	提取字符，包括空格	1. int get();提取一个字符； 2.get(char *pch,int ncount,char delim='\n');遇到分隔符delim后结束
getline	提取一行字符	getline(char *pch,int ncount,char delim='\n')； 若前有输入cin>>a,要先添加cin.get()后再写cin.getline(……)；

3.输出流操作

函数	功能
put ,write	插入一字节
seekp	移动输出流指针
tellp	返回输出流中指定位置的指针值

4 恢复或设置状态字cin.clear();或者cin.clear(ios::goodbit);

```
while(cin>>k)        //按ctrl_z结束输入
{
    total+=k;
}
cin.clear();           //状态字清零，恢复流状态
while(cin>>k)
{
    total-=k;
}
```

3.格式控制

标志常量	意义
ios::left	左对齐
ios::right	右对齐
ios::showbase	在输出中显示基数指示符
ios::showpoint	显示小数点
ios::showpos	正数显示+
ios::scientific	科学计数法显示浮点数
ios::fixed	用定点形式显示浮点数

函数	功能
int width(int n)	设置和返回输出宽度 cin.width(10);自动右对齐，输出后将恢复系统默认设置
char fill(char c)	设置填充字符
int precision(int n)	设置显示精度

cout.setf(ios::oct,ios::basefield);清除基数标志位

可以用 | 同时设置几个标志字体

4.iostream常用的控制符

```
cout<< hex<<a;   //十六进制
cout<<oct<<a<<endl;
cout<<dec<<a<<endl;
```

5.iomanip常用的控制符

控制符	**功能**
resetiosflags(ios::iflags)	清除iflags指定的标志位
setiosflags(ios::iflags)	设置iflags指定的标志位
setbase(int base)	设置基数
setfill(char c)	
setprecision(int n)	设置浮点数输出宽度
setw(int n)	设置输出宽度，只对后面紧跟着的数据起作用