C++第四讲

一、友元 (friend)

功能:允许朋友访问自己的所有权限下的成员,包括私有成员。

1.1 友元函数

1.1.1 全局函数作为友元函数

声明一个全局函数作为类的友元函数,需要在类中进行声明: friend 函数头;

```
1 #include <iostream>
 2 class Stu;
                       //声明类
   using namespace std;
 4
 5 class Stu
 6 {
 7
   private:
 8
        string name;
9
       int age;
        double score;
10
11
   public:
12
13
        Stu() {}
        Stu(string n, int a, double s):name(n),age(a),score(s) {}
14
15
       ~Stu() {}
16
17
      void show()
18
            cout<<"name = "<<name<<endl;</pre>
19
20
            cout<<"age = "<<age<<endl;</pre>
            cout<<"score = "<<score<<endl<<endl;</pre>
21
22
        }
23
        friend void fun(Stu &s);
                                             //声明fun函数是我的朋友
24
25
   };
26
27
   //定义全局函数输出信息
28 void display(Stu &s)
29
```

```
30 | // cout<<"name = "<<s.name<<end1;</pre>
  // cout<<"age = "<<s.age<<endl;</pre>
31
   //
        cout<<"score = "<<s.score<<endl<<endl;</pre>
32
33
      s.show();
   }
34
35
   //定义全局函数改变信息并输出
36
37
   void fun(Stu &s)
38
   {
39
      s.name = "lisi";
40
41
       s.age = 20;
42
43
        cout<<"name = "<<s.name<<endl;</pre>
44
        cout<<"age = "<<s.age<<endl;</pre>
        cout<<"score = "<<s.score<<endl<<endl;</pre>
45
46
   }
47
48
49
   int main()
50
   {
51
        Stu s1("zhangpp", 18, 99);
52
        s1.show();
                                     //直接通过类对象调用成员函数
53
        display(s1); //调用全局函数实现输出
54
55
56
        fun(s1);
57
58
        return 0;
59
   }
60
```

练习: 定义两个类: 狗类 (Dog) 、猴子类 (Monkey) , 分别有成员: 个数、重量

定义一个全局函数,统计出狗和猴子的总个数,和总重量

```
#include <iostream>

using namespace std;

class monkey; //类的前置声明

class dog

{
private:
   int num;
   float weight;
```

```
public:
10
11
       dog() {}
       dog(int n,float w):num(n),weight(w) {}
12
       friend void total(dog a, monkey b);
                                                   //声明全局函数
13
   作为友元函数
14
   };
15 class monkey
16 {
   private:
17
       int num;
18
19
       float weight;
   public:
20
       monkey() {}
21
       monkey(int n,float w):num(n),weight(w) {}
22
23
       friend void total(dog a, monkey b); //声明全局函数作为
   友元函数
24
   };
25 void total(dog a, monkey b)
26
       cout<<"总个数:"<<a.num+b.num<<endl;
27
       cout<<"总重量:"<<a.weight+b.weight<<endl;
28
29
30
   int main()
31
   {
       dog d1(10,50.2);
32
33
       monkey m1(8, 36.5);
34
       total(d1,m1);
35
      return 0;
   }
36
```

1.1.2 类的成员函数作为友元函数 (了解)

声明一个其他类的函数作为友元函数,但是需要在那个类中进行声明,类外定义,允许其他类的成员函数访问

1.2 友元类

在类中将另一个类作为友元类,那么,这个友元类中的任何成员都可以访问该 类的成员

```
1 #include <iostream>
2
3 using namespace std;
4 class Stu; //声明学生类
5 class Teacher;
```

```
7
   class Teacher
8
    {
9
   private:
        string t_name;
10
        int t_age;
11
12
13
    public:
        Teacher() {}
14
15
        Teacher(string n, int a):t_name(n), t_age(a) {}
        ~Teacher() {}
16
17
        void show()
18
19
        {
20
            cout<<"t_name = "<<t_name<<endl;</pre>
            cout<<"t_age = "<<t_age<<endl;</pre>
21
22
        }
23
24
        void show(Stu s);
                                       //类内声明
25
        //friend class Stu;
26
27
28
   };
29
30
31 class Stu
32
    {
33 private:
34
       string name;
35
       int age;
        double score;
36
37
38
    public:
39
        Stu() {}
        Stu(string n, int a, double s):name(n),age(a),score(s) {}
40
        ~Stu() {}
41
42
43
        void show()
        {
44
            cout<<"name = "<<name<<end1;</pre>
45
            cout<<"age = "<<age<<endl;</pre>
46
            cout<<"score = "<<score<<endl<<endl;</pre>
47
48
        }
49
        friend void fun(Stu &s);
                                              //声明fun函数是我的朋友
50
```

```
51 //friend void Teacher::show(Stu s); //声明老师类中的
   函数为自己的友元函数
       friend class Teacher;
                                           //声明整个类作为友元类,
52
   允许teacher访问自己私有成员
   };
53
54
55 //定义全局函数输出信息
56 | void display(Stu &s)
57 {
58 // cout<<"name = "<<s.name<<endl;
59 // cout<<"age = "<<s.age<<endl;</pre>
60 // cout<<"score = "<<s.score<<endl<;
61 s.show();
62 }
63
64 //定义全局函数改变信息并输出
65 void fun(Stu &s)
66 {
67
68
       s.name = "lisi";
69
       s.age = 20;
70
71
       cout<<"name = "<<s.name<<endl;</pre>
72
       cout<<"age = "<<s.age<<endl;</pre>
73
       cout<<"score = "<<s.score<<endl<<endl;</pre>
74
   }
75
76
77 | void Teacher::show(Stu s) //teacher类中show函数的定义
78
   {
79
           cout<<"name = "<<s.name<<endl;</pre>
           cout<<"age = "<<s.age<<endl;</pre>
80
81
           cout<<"score = "<<s.score<<endl<<endl;</pre>
82
   }
83
84
85 | int main()
86 {
       Stu s1("zhangpp", 18, 99);
87
88
       Teacher t1("zhangsan", 30);
89
       t1.show();
90
91
       t1.show(s1);
                            //输出学生的信息
92
93
```

```
94 return 0;
95 }
96
```

1.3 使用友元的注意事项

1> 友元不具有交换性: A是B的朋友, B不一定是A的朋友

2> 友元不具有传递性: A是B的朋友, B是C的朋友, A不一定是C的朋友

3> 友元不具有继承性: 父类的朋友不一定是子类的朋友

4> 尽可能少的使用友元: 因为友元破坏了类的封装性, 使得封装徒有其表, 所

以,不在玩不得已的情况下,不要使用友元

5> 需要使用友元的情况:插入和提取运算符重载时,需要使用友元来实现

二、常成员函数和常对象 (const)

2.1 常成员函数

1> 格式: 定义成员函数后加关键字const 返回值类型 函数名(参数列表) const

2> 作用:在常成员函数中,不允许修改成员变量的值,保护成员变量不被修改

3> this的原型: const 类名 * const this; --->既不能改变指向, 也不能通过 this指针改变值

非常成员函数this的原型: 类名 * const this; --->不能改变指向, 但是可以通过this指针改变值

- 4> 同名的长成员函数和非常成员函数构成重载关系
- 5> 非常对象,优先调用非常成员函数,如果没有非常成员函数,那么会调用同名的常成员函数

2.2 常对象

- 1> 格式:在实例化对象前加const修饰后,那么该对象便是常对象const 类名 对象名;
- 2> 作用: 常对象只能调用常成员函数,不能调用非常成员函数
- 3> 当常引用的目标为非常对象时,如果之前没有调用过同名函数,通过常引用调用的是常成员函数,通过对象名调用的是非常成员函数

```
#include <iostream>

using namespace std;

class Stu
{
private:
```

```
string name;
9
      int age;
10
11 public:
12
      Stu() {}
13
      Stu(string n, int a):name(n), age(a) {}
   ~Stu() {}
14
15
16
17
     void show() const //const 类名 * const this
18
     {
        //age = 100; //在常成员函数中,是不可以对成员变量
19
   进行修改的
20
      cout<<name<<endl;</pre>
21
22
     cout<<this->age<<endl;
23
     }
24
     void show()  //类名 * const this;
25
     {
26
         this->age = 100; //在非 常成员函数中,是可以对成
27
   员变量进行修改的
28
        cout<<name<<endl;</pre>
29
30
        cout<<age<<end1;
31
     }
32 };
33
34
35 int main()
36 {
37
     Stu s1("zhangpp", 18);
38
   //s1.show();
                             //100
39
40
     const Stu &r = s1;
                            //定义常引用,目标为非常对象
41
     r.show();
                           // 18
42
43
      s1.show();
                           //100
44
45
     return 0;
46 }
47
```

2.3 mutable关键字

```
1 #include <iostream>
2
  using namespace std;
3
4
  class Stu
5
6 {
7
  private:
8
      string name;
      mutable int age; //该成员变量可以在常成员函数中被修改
9
10
11 public:
12
      Stu() {}
      Stu(string n, int a):name(n), age(a) {}
13
    ~Stu() {}
14
15
16
    void show() const //const 类名 * const this
17
18
          this->age = 200;
                                  //在常成员函数中,是不可以对成员
19
   变量进行修改的
          //this->name = "王勃"; //该成员不能被修改,因为没有
20
   mutable修饰
21
22
          cout<<name<<end1;</pre>
23
          cout<<this->age<<endl;</pre>
24
     }
25
    void show()  //类名 * const this;
26
27
      {
          this->age = 100; //在非 常成员函数中,是可以对成
28
   员变量进行修改的
29
30
          cout<<name<<end1;</pre>
         cout<<age<<end1;</pre>
31
32
       }
33
34
   };
35
36
37 int main()
38 {
      Stu s1("zhangpp", 18);
39
```

```
//s1.show();
40
                               //100
41
      const Stu &r = s1;
42
                            //定义常引用,目标为非常对象
43
      r.show();
                            // 18
     s1.show();
                            //100
44
45
46 return 0;
47 | }
```

三、运算符重载

单、算、关、逻、条、赋、逗、位

3.1 定义

所谓运算符重载,能够实现"一符多用",其本质也还是运算符重载的一种,属于静态多态,可以通过运算符重载实现自定义类对象之间的运算,使得代码更加简便、 优雅

3.2 重载的方法

统一名称: operator # (#表示运算符的名字)

3.3 运算符重载参数

参数个数由运算符本身决定,返回值类型可以由程序员决定

3.4 调用时机及调用原则

调用时机: 当使用该运算符时, 系统自动调用重载函数

调用原则: 左调右参 例如: a = b; ----> a.operator=(b)

3.5 运算符重载的格式

统一格式:返回值类型 operator # (形参表)

每种运算符都有两个版本的重载函数:

成员函数版: 全局函数版:

实际使用时,只能实现一个版本的情况,通常实现成员函数版

3.6 算数类运算符重载

1> +, -, *, /, %. . .

2> 表达式: L#R (L表示左操作数、R表示右操作数、#表示运算符)

3> 左操作数: 既可以是左值也可以是右值

4> 右操作数: 既可以是左值也可以是右值

5> 结果: 只能是右值

6> 格式:

成员函数版: const 类名 operator#(const 类名& R) const;

左边第一个const:表示结果是个右值,不能被修改

左边第二个const:表示右操作数在运算过程中不能被修改 左边第三个const:表示左操作数在运算过程中不能被修改

全局函数版: const 类名 operator# (const 类名& L, const 类名&

R)

3.7 赋值类运算符重载

1> 种类:

基本赋值运算:=

复合赋值运算: +=、-=、*=、/=、%=。。。。

2> 表达式: L#R (L表示左操作数、R表示右操作数、#表示运算符)

3> 左操作数: 只能是左值

4> 右操作数: 既可以是左值也可以是右值

5> 结果: 左自身的引用

6> 格式:

成员函数版: 类名 & operator#(const 类名& R);

全局函数版: 类名 & operator# (类名& L, const 类名& R)

3.8 关系运算符重载

1> 种类: > < == != >= <=

2> 表达式:L#R (L表示左操作数、R表示右操作数、#表示运算符)

3> 左操作数: 既可以是左值也可以是右值

4> 右操作数: 既可以是左值也可以是右值

5> 结果: bool 右值

6> 格式:

成员函数版: const bool operator#(const 类名& R)const; 全局函数版: bool operator# (const 类名& L, const 类名& R)

3.9 单目运算符重载

1> 种类:!、~、[]、-(负号)

2> 表达式: #O (O表示左操作数、#表示运算符)

3> 操作数: 既可以是左值也可以是右值

5> 结果: 只能是右值

6> 格式:

成员函数版: const 类名 operator#()const;

全局函数版: const 类名 operator# (const 类名& O)

3.10 自增、自减运算符重载

```
1 前置自增: ++a
```

1> 表达式: #O

2> 操作数: 只能是左值 3> 结果: 自身的引用

4> 格式:

成员函数版: 类名 & operator#();

全局函数版: 类名 & operator# (类名& O)

前置自增: a++ 1> 表达式: #O

2> 操作数: 只能是左值

3> 结果: 右值

4> 格式:

成员函数版: 类名 operator#(int);

全局函数版: 类名 operator# (类名& O, int)

```
#include <iostream>
 1
 2
 3
   using namespace std;
 4
 5
   class Complax
 6
 7
   private:
       int rea; //实部
 8
9
       int vir;
                    //虚部
10
   public:
11
12
       Complax() {}
13
       Complax(int r, int v):rea(r), vir(v) {}
14
```

```
15
       void show()
       {
16
17
           cout<<rea<<" + "<<vir<<"i"<<endl;</pre>
18
       }
19
20
       //实现+运算符重载 : 实部+实部 虚部+虚部
21
       const Complax operator+(const Complax& R ) const
22
       {
23
           Complax temp;
24
25
           temp.rea = this->rea+R.rea;
           temp.vir = this->vir+R.vir;
26
27
28
           //R.rea = 100;
29
30
           return temp;
       }
31
32
33
       friend const Complax operator-(const Complax& L, const
                  //声明全局函数为友元
   Complax& R);
34
       friend Complax & operator-=(Complax &L ,const Complax& R
   );
35
       //实现+=运算符重载: 实部 +=实部 虚部+=虚部
36
37
       Complax & operator+=(const Complax& R )
38
       {
39
           this->rea += R.rea;
40
           this->vir += R.vir;
41
42
           return *this;
43
       }
44
       //实现>运算符重载 : 实部>实部 虚部>虚部
45
46
       bool operator>(const Complax& R )const
47
       {
           return this->rea>R.rea&&this->vir>R.vir;
48
49
       }
50
       //实现负号运算符重载:实部 = -实部 虚部=-虚部
51
       const Complax operator-()const
52
53
       {
54
           Complax temp;
55
56
           temp.rea = -this->rea;
           temp.vir = -this->vir;
57
```

```
58
 59
            return temp;
 60
        }
61
62
        //实现前置自增运算符重载:实部 = 实部+1 虚部=虚部+1
        Complax & operator++()
63
64
        {
65
            this->rea++;
 66
            this->vir++;
67
68
            return *this;
69
        }
70
        //实现后置自增运算符重载:实部 = 实部+1 虚部=虚部+1
71
72
        Complax operator++(int)
73
        {
74
            Complax temp;
75
76
            temp.rea = this->rea++;
77
            temp.vir = this->vir++;
 78
 79
            return temp;
 80
        }
 81
82
83
84
    };
85
    const Complax operator-(const Complax& L, const Complax& R)
 86
 87
    {
 88
        Complax temp;
89
90
        temp.rea = L.rea-R.rea;
        temp.vir = L.vir-R.vir;
91
92
93
        return temp;
94
    }
95
    //全局函数版 -=运算符重载
96
97
    Complax & operator==(Complax &L ,const Complax& R )
98
99
        L.rea -= R.rea;
100
        L.vir -= R.vir;
101
102
        return L;
```

```
103
104
105
106
107
    int main()
108
109
         Complax c1(3,5);
         c1.show();
110
                              //3+5i
111
112
        Complax c2(2,4);
113
         c2.show();
                             //2+4i
114
115
        Complax c3;
116
117
         c3 = c1+c2; //c1.operator(c2)
118
         c3.show();
                                //5+9i
119
120
         c3 = c1-c2;
121
                               //1+1i
         c3.show();
122
         c3 += c1; //实现了+=运算符重载
123
124
         c3.show();
                                //4+6i
125
126
         c3-=c1;
127
         c3.show(); //1+1i
128
129
        if(c1>c3)
130
         {
             cout<<"yes"<<endl;</pre>
131
132
        }else
133
         {
134
             cout<<"no"<<endl;</pre>
135
         }
136
137
         c3 = -c1;
                           //-3 + -5i
138
         c3.show();
139
140
        ++c1;
                          //
                               //4+6i
141
         c1.show();
142
143
         Complax c4 = c1++;
144
145
         c4.show();
                               //4+6i
146
         c1.show();
                               //5+7i
147
```

```
148
149
150
151 return 0;
152 }
153
```

3.11 插入、提取运算符重载

1> cin和cout的来源

```
1 namespace std
2 {
3    istream cin;
4    ostream cout;
5 }
```

- 2> 想要对插入运算符 (<<) 和提取运算符(>>)进行重载,需要对istream和 ostream类进行修改,难度较大
- 3> 此时,我们可以实现全局函数版,将其作为自定义类的友元函数来实现对自定义类的访问
- 4> 格式:

表达式: L#R //L是cin或者cout #表示插入 (<<) 或者提取

(>>) 运算符

左操作数: istream或ostream的类对象

右操作数: 自定义类对象 结果: 左操作数自身的引用

5> 全局函数版:

ostream &operator<<(ostream &L, const 自定义类 &R); //插入

运算符重载格式

istream & operator>>(istream &L, const 自定义类 &R); //插入运算符重载格式

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Complax
6 {
7 private:
8 int rea; //实部
9 int vir; //虚部
```

```
10
   public:
11
12
       Complax() {}
13
       Complax(int r, int v):rea(r), vir(v) {}
14
15
       void show()
16
       {
17
           cout<<rea<<" + "<<vir<<"i"<<endl;</pre>
18
       }
19
20
       //实现+运算符重载 : 实部+实部
                                     虚部+虚部
       const Complax operator+(const Complax& R ) const
21
22
       {
23
           Complax temp;
24
25
           temp.rea = this->rea+R.rea;
           temp.vir = this->vir+R.vir;
26
27
28
           //R.rea = 100;
29
30
           return temp;
31
       }
32
       friend const Complax operator-(const Complax& L, const
33
   Complax& R);
                   //声明全局函数为友元
34
       friend Complax & operator-=(Complax &L ,const Complax& R
   );
35
       friend ostream &operator<<(ostream &L, const Complax &R);</pre>
           //将插入函数设为友元函数
36
37
       //实现+=运算符重载: 实部 +=实部 虚部+=虚部
38
       Complax & operator+=(const Complax& R )
39
       {
40
           this->rea += R.rea;
41
           this->vir += R.vir;
42
43
           return *this;
44
       }
45
       //实现>运算符重载 : 实部>实部
46
                                  虚部>虚部
47
        bool operator>(const Complax& R )const
48
       {
49
           return this->rea>R.rea&&this->vir>R.vir;
50
       }
51
```

```
//实现负号运算符重载:实部 = -实部 虚部=-虚部
52
53
       const Complax operator-()const
54
       {
           Complax temp;
55
56
           temp.rea = -this->rea;
57
58
           temp.vir = -this->vir;
59
60
           return temp;
61
       }
62
63
       //实现前置自增运算符重载:实部 = 实部+1 虚部=虚部+1
64
       Complax & operator++()
65
       {
66
           this->rea++;
67
           this->vir++;
68
69
           return *this;
70
       }
71
72
       //实现后置自增运算符重载:实部 = 实部+1 虚部=虚部+1
73
       Complax operator++(int)
74
       {
75
           Complax temp;
76
77
           temp.rea = this->rea++;
78
           temp.vir = this->vir++;
79
80
           return temp;
       }
81
82
83
84
85
   };
86
87
   const Complax operator-(const Complax& L, const Complax& R)
   {
88
89
       Complax temp;
90
91
       temp.rea = L.rea-R.rea;
92
       temp.vir = L.vir-R.vir;
93
94
       return temp;
95
   }
96
```

```
97 //全局函数版 -=运算符重载
    Complax & operator-=(Complax &L ,const Complax& R )
 98
 99
100
        L.rea -= R.rea;
101
        L.vir -= R.vir;
102
103
       return L;
104
    }
105
106
    //全局函数版实现插入运算符重载函数
    ostream &operator<<(ostream &cout, const Complax &R)
107
108
109
        cout << R.rea<<" + "<<R.vir<<"i";</pre>
110
111
        return cout;
112
    }
113
114
115
116
    int main()
117
118
        Complax c1(3,5);
119
        c1.show();
                             //3+5i
120
121
        Complax c2(2,4);
122
        c2.show();
                            //2+4i
123
124
        Complax c3;
125
        c3 = c1+c2; //c1.operator(c2)
126
127
                               //5+9i
        c3.show();
128
129
        c3 = c1-c2;
130
        c3.show();
                               //1+1i
131
132
        c3 += c1; //实现了+=运算符重载
133
        c3.show();
                               //4+6i
134
135
        c3-=c1;
                     //1+1i
136
        c3.show();
137
138
        if(c1>c3)
139
140
            cout<<"yes"<<endl;</pre>
141
        }else
```

```
142
           cout<<"no"<<endl;
143
144
        }
145
146
147
148
       c3 = -c1;
        c3.show();
                         //-3 + -5i
149
150
151
       ++c1;
                      //
                            //4+6i
152
       c1.show();
153
154
      Complax c4 = c1++;
155
156
      c4.show();
                          //4+6i
157
      c1.show();
                           //5+7i
158
159
     cout<<c1<<end1<<c2<<end1; //cout.operator<<(c1)</pre>
160
161 return 0;
162 }
```

3.12 不能重载的运算符

- 1> 成员运算符.
- 2> 成员指针运算符.*
- 3> 作用域限定符::
- 4> 条件表达式?:
- 5> 求字节运算 sizeof()

3.13 运算符重载注意事项

- 1> 运算符重载不能改变运算符的本质:例如不能在加法运算中实现减法功能
- 2> 运算符重载,只能在已有运算符上进行重载,不能自己造运算符:例如不能 **表示乘方运算
 - 3> 运算符重载不能改变运算符的优先级
 - 4> 运算符重载不能改变运算符的参数:如双目运算符参数最多两个
 - 5> 运算符重载不能改变运算符的结合律
 - 6> 运算符重载不能改变运算符的默认参数

作业

继续完成my_string 类的书写,主要完成运算符重载:

+=运算符: operator+=

下标运算符: operator[] 加法运算符: operator+

关系运算符: > 、 <、==

要求: 自己分析函数, 该加const的要准确加上