

Assignment 9

191300051¹

Wang Fuyun¹

1.概念简答题

1.1 以下关于函数模板的描述是否有错误？如果有错误请指出，并给出理由。

1. 函数模板必须由程序员实例化为可执行的函数模板
2. 函数模板的实例化由编译器实现
3. 一个类定义中，只要有一个函数模板，则这个类是类模板
4. 类模板的成员函数都是函数模板，类模板实例化后，成员函数也随之实例化

答：

1, 3, 4都是错误的。

1. 函数模板必须由编译器根据程序员的调用类型实例化为可执行的函数。
2. 正确。函数模板的实例化由编译器实现。也就是说在编译得到的.i文件中会看到多个能够处理多种变量类型的成员函数。
3. 类模板的成员函数都是函数模板。而不是只要有一个。
4. 没使用过的成员函数（即函数模板）不会被实例化。

1.2. 在类模版中使用静态成员有什么要注意的地方？

不同类模板实例之间，不共享类模板实例的静态成员。

1.3. 为什么通常情况下，类模板的定义和实现都放在头文件中？

因为模板的实例化发生在编译阶段或者说预处理阶段的，而预处理阶段是每个模块分开进行的，如果只放在某个模块当中，那么预处理的时候只会检查该模块当中进行了哪些模板调用，并进行相关的实例化。此时如果在其他模块当中，声明了该模板，但是没有调用了之前没有实例化的参数类型的函数，那么就会导致找不到相应的函数的问题。

1.4 关于typename和class的一点区别

又是历史原因，以前是用class，后来C++ Standard 出现后，引入了typename，所以他们是一样的。

但是，又有一些微妙的不同，因为有时候，你不得不使用typename.

1. 在声明 template parameters (模板参数) 时，class 和 typename 是可互换的。
2. 用 typename 去标识 nested dependent type names (嵌套依赖类型名)，在 base class lists (基类列表) 中或在一个 member initialization list (成员初始化列表) 中作为一个 base class identifier (基类标识符) 时除外。

2.编程题

2.1 用模板类实现一个最大堆

```
1  #include <iostream>
2  #include <algorithm>
3  using namespace std;
4
5  //实现能够有多种数据类型的最大堆。
6  template <class Type>
7  class MaxHeap
8  {
9  private:
10     Type *data;
11     int size;    //当前大小
12     int capacity; //总容量
13     int parent(int i)
14     {
15         // -1/2 = -1;
16         return (i - 1) / 2;
17     }
18     int left(int i)
19     {
20         return i * 2 + 1;
21     }
22     int right(int i)
23     {
24         return (i + 1) * 2;
25     }
26     void heapify(int i)
27     {
28         int l = left(i);
29         int r = right(i);
30         int largest = i;
31         if (l < size && data[l] > data[largest])
```

```

32     {
33         largest = 1;
34     }
35     if (r < size && data[r] > data[largest])
36     {
37         largest = r;
38     }
39     if (largest != i)
40     {
41         swap(data[i], data[largest]);
42         heapify(largest);
43     }
44 }
45
46 public:
47     MaxHeap(int c = 10) : data(new Type[c]), size(0), capacity(c)
48     {
49     }
50     //直接从数组进行建堆
51     //则指针A指向的数组不可变。
52     MaxHeap(const Type A[], int n, int c) : data(new Type[c]), size(n), capacity(c)
53     {
54         for (int i = 0; i < n; i++)
55         {
56             data[i] = A[i];
57         }
58         for (int i = (size - 1) / 2; i >= 0; i--)
59         {
60             heapify(i);
61         }
62     }
63     //默认的析构函数不会删除掉动态变量。
64     ~MaxHeap()
65     {
66         delete data;
67     }
68     bool Insert(Type element) //插入一个元素
69     {
70         data[size] = element;
71         int i = size;
72         size++;
73         while (i >= 0 && data[parent(i)] < data[i])
74         {
75             swap(data[parent(i)], data[i]);
76             i = parent(i);
77         }
78     }
79     //让出0位置的元素，将末尾元素置于0位置，并执行heapify。
80     Type DeleteMax() //找出最大的元素返回，并进行删除
81     {
82         Type ans = data[0];
83         size--;
84         data[0] = data[size];
85         heapify(0);
86         return ans;
87     }
88     bool IsFull() //是否为满
89     {
90         return size == capacity;
91     }
92     bool IsEmpty() //是否为空

```

```

93     {
94         return size == 0;
95     }
96     void Print() //打印
97     {
98         cout << "PRINT THE ELEMENT IN HEAP\n";
99         for (int i = 0; i < size; i++)
100         {
101             cout << data[i] << " ";
102             if (left(i) < size)
103             {
104                 cout << data[left(i)]<<" ";
105             }
106             if (right(i)<size)
107             {
108                 cout<<data[right(i)]<<" ";
109             }
110             cout<<endl;
111         }
112     }
113 };
114
115 int main()
116 {
117     cout << "TEST1\n";
118     int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
119     MaxHeap<int> h1(a, 10, 20);
120     h1.Print();
121     h1.Insert(11);
122     h1.Print();
123     h1.Insert(100);
124     h1.Insert(1000);
125     cout << h1.DeleteMax() << endl;
126     h1.Print();
127
128     cout << "TEST2\n";
129     MaxHeap<int> h2(20);
130     for (int i = 1; i < 11; i++)
131     {
132         h2.Insert(i);
133     }
134     h2.Print();
135     h2.Insert(11);
136     h2.Print();
137     h2.Insert(100);
138     h2.Insert(1000);
139     cout << h2.DeleteMax() << endl;
140     h2.Print();
141 }

```

最终输出结果为

每行输出父节点以及如果存在子节点则输出子节点。

```

1  TEST1
2  PRINT THE ELEMENT IN HEAP
3  10 9 7
4  9 8 5
5  7 6 3
6  8 1 4

```

```
7 5 2
8 6
9 3
10 1
11 4
12 2
13 PRINT THE ELEMENT IN HEAP
14 11 10 7
15 10 8 9
16 7 6 3
17 8 1 4
18 9 2 5
19 6
20 3
21 1
22 4
23 2
24 5
25 1000
26 PRINT THE ELEMENT IN HEAP
27 100 10 11
28 10 8 9
29 11 7 3
30 8 1 4
31 9 2 5
32 7 6
33 3
34 1
35 4
36 2
37 5
38 6
39 TEST2
40 PRINT THE ELEMENT IN HEAP
41 10 9 6
42 9 7 8
43 6 2 5
44 7 1 4
45 8 3
46 2
47 5
48 1
49 4
50 3
51 PRINT THE ELEMENT IN HEAP
52 11 10 6
53 10 7 9
54 6 2 5
55 7 1 4
56 9 3 8
57 2
58 5
59 1
60 4
61 3
62 100 10 11
63 10 7 9
64 11 6 5
65 7 1 4
66 9 3 8
67 6 2
```

```
68 | 5
69 | 1
70 | 4
71 | 3
72 | 8
73 | 2
74 | P
```

2.2在第五次作业中，我们实现了一个基于int类型的Matrix类，这次作业我们对这个类进行一次扩展，希望扩展后它能处理多种数据类型的矩阵运算，这些数据类型包括int 和复数类

```
1 //实现支持多种数据类型，包括自建的复数类的矩阵类
2 #include <iostream>
3 #include <cstdio>
4 #include <cstring>
5 #include <cassert>
6 using namespace std;
7
8 //在Complex类中，可以直接访问其他Complex对象的private资源。
9 class Complex
10 {
11 private:
12     double real, imag;
13
14 public:
15     Complex(double r = 0, double i = 0) : real(r), imag(i)
16     {
17     }
18     void print()
19     {
20         if (imag == 0)
21         {
22             cout << real << endl;
23         }
24         else if (imag > 0)
25         {
26             cout << real << "+" << imag << "j" << endl;
27         }
28         else
29         {
30             cout << real << imag << "j" << endl;
31         }
32     }
33     double modulus() const
34     {
35         return real * real + imag * imag;
36     }
37     //默认在左边的符号，如果想要在右边，则传入一个int参数进来
38     //这里因为不是一个赋值操作所以必须返回的是值，也可以是动态变量的指针或者引用
39     //但是如果没指针接受它的话则内存泄漏
40     //另外一定不能用静态的临时变量，返回指针或者引用，因为函数结束以后静态变量就已经被释放了。
41     //想要在左边就能用，必须写成类内的重载函数了。
```

```

42     Complex operator-() const
43     {
44         Complex tmp;
45         tmp.real = -real;
46         tmp.imag = -imag;
47         return tmp;
48     }
49     //=-号的重载也必须是成员函数，与拷贝构造函数基本一致。
50     Complex &operator=(const Complex &c)
51     {
52         real = c.real;
53         imag = c.imag;
54         return *this;
55     }
56     //但是一般这些我们都会传入const的引用，既可以避免修改原值，又可以加快速度。
57     friend bool operator==(const Complex &c1, const Complex &c2);
58     friend bool operator!=(const Complex &c1, const Complex &c2);
59     friend bool operator>(const Complex &c1, const Complex &c2);
60     friend bool operator>=(const Complex &c1, const Complex &c2);
61     friend bool operator<(const Complex &c1, const Complex &c2);
62     friend bool operator<=(const Complex &c1, const Complex &c2);
63     friend Complex operator+(const Complex &c1, const Complex &c2);
64     friend Complex operator-(const Complex &c1, const Complex &c2);
65     friend Complex operator*(const Complex &c1, const Complex &c2);
66     friend Complex operator/(const Complex &c1, const Complex &c2);
67     friend ostream &operator<<(ostream &output, const Complex &c);
68 };
69 bool operator==(const Complex &c1, const Complex &c2)
70 {
71     return c1.real == c2.real && c1.imag == c2.imag;
72 }
73 bool operator!=(const Complex &c1, const Complex &c2)
74 {
75     return !(c1 == c2);
76 }
77 bool operator>(const Complex &c1, const Complex &c2)
78 {
79     return c1.modulus() > c2.modulus();
80 }
81 bool operator>=(const Complex &c1, const Complex &c2)
82 {
83     return c1.modulus() >= c2.modulus();
84 }
85 bool operator<(const Complex &c1, const Complex &c2)
86 {
87     return c1.modulus() < c2.modulus();
88 }
89 bool operator<=(const Complex &c1, const Complex &c2)
90 {
91     return c1.modulus() <= c2.modulus();
92 }
93 Complex operator+(const Complex &c1, const Complex &c2)
94 {
95     return Complex(c1.real + c2.real, c1.imag + c2.imag);
96 }
97 Complex operator-(const Complex &c1, const Complex &c2)
98 {
99     return Complex(c1.real - c2.real, c1.imag - c2.imag);
100 }
101 Complex operator*(const Complex &c1, const Complex &c2)
102 {

```

```

103     return Complex(c1.real * c2.real - c1.imag * c2.imag, c1.real * c2.imag + c1.imag *
104     c2.real);
105 }
106 Complex operator/(const Complex &c1, const Complex &c2)
107 {
108     double d = c2.modulus();
109     if (d == 0)
110     {
111         cerr << "Error in operation / of Complex" << endl;
112         exit(-1);
113     }
114     else
115     {
116         return Complex((c1.real * c2.real + c1.imag * c2.imag) / d, (c1.imag * c2.real -
117         c1.real * c2.imag) / d);
118     }
119 }
120 ostream &operator<<(ostream &output, const Complex &c)
121 {
122     int num = c.imag;
123     if (num >= 0)
124     {
125         output << c.real << "+" << c.imag << "i";
126     }
127     else
128     {
129         output << c.real << c.imag << "i";
130     }
131     return output;
132 }
133
134 void testComplex()
135 {
136     Complex a(1, 1), b(2, 2), c(0, 0);
137     a.print();
138     b.print();
139     c.print();
140     cout << "a<b:" << (a < b) << endl;
141     (a * b).print();
142     (a / b).print();
143     (a + b).print();
144     (a * c).print();
145
146     //divided by zero exit(-1);
147     // (a / c).print();
148 }
149
150 template <class T>
151 class Matrix
152 {
153 private:
154     int colNum;
155     int rowNum;
156     T *data;
157
158 public:
159     Matrix(int r = 0, int c = 0)
160     {
161         colNum = c;

```



```

162     rowNum = r;
163     data = new T[colNum * rowNum];
164     // for (int i=0;i<r*c;i++)
165     // {
166     //
167     // }
168     memset(data, 0, sizeof(T) * c * r);
169 }
170 Matrix(int r, int c, T a[])
171 {
172     colNum = c;
173     rowNum = r;
174     data = new T[colNum * rowNum];
175     for (int i = 0; i < r * c; i++)
176     {
177         data[i] = a[i];
178     }
179 }
180 void print()
181 {
182     for (int i = 0; i < rowNum; i++)
183     {
184         for (int j = 0; j < colNum; j++)
185         {
186             cout << data[i * colNum + j] << " ";
187         }
188         cout << endl;
189     }
190 }
191 //赋值操作
192 Matrix &operator=(const Matrix &m)
193 {
194     colNum = m.colNum;
195     rowNum = m.rowNum;
196     delete data;
197     data = new T[colNum * rowNum];
198     memset(data, m.data, sizeof(T) * colNum * rowNum);
199     return *this;
200 }
201
202 //实现矩阵类
203 Matrix operator-()
204 {
205     Matrix m(rowNum, colNum);
206     for (int i = 0; i < colNum * rowNum; i++)
207     {
208         m[i] = -data[i];
209     }
210 }
211 T *operator[](int index)
212 {
213     return &data[index * colNum];
214 }
215 //函数模板的声明上面也要加!
216 //友元函数的class和模板类的class还不能一样?
217 template <class Ty>
218 friend Matrix<Ty> operator+(const Matrix<Ty> &m1, const Matrix<Ty> &m2);
219 template <class Ty>
220 friend Matrix<Ty> operator-(const Matrix<Ty> &m1, const Matrix<Ty> &m2);
221 template <class Ty>
222 friend Matrix<Ty> operator*(const Matrix<Ty> &m1, const Matrix<Ty> &m2);

```

```

223 };
224 template <class T>
225 Matrix<T> operator+(const Matrix<T> &m1, const Matrix<T> &m2)
226 {
227     assert(m1.colNum == m2.colNum && m1.rowNum == m2.rowNum);
228     Matrix<T> tmp(m1.rowNum, m1.colNum);
229     for (int i = 0; i < tmp.colNum * tmp.rowNum; i++)
230     {
231         tmp.data[i] = m1.data[i] + m2.data[i];
232     }
233     return tmp;
234 }
235 template <class T>
236 Matrix<T> operator-(const Matrix<T> &m1, const Matrix<T> &m2)
237 {
238     assert(m1.colNum == m2.colNum && m1.rowNum == m2.rowNum);
239     Matrix<T> tmp(m1.rowNum, m1.colNum);
240     for (int i = 0; i < tmp.colNum * tmp.rowNum; i++)
241     {
242         tmp.data[i] = m1.data[i] - m2.data[i];
243     }
244     return tmp;
245 }
246 template <class T>
247 Matrix<T> operator*(const Matrix<T> &m1, const Matrix<T> &m2)
248 {
249     assert(m1.colNum == m2.rowNum);
250     Matrix<T> tmp(m1.rowNum, m2.colNum);
251     for (int i = 0; i < tmp.rowNum * tmp.colNum; i++)
252     {
253         int r = i / tmp.colNum;
254         int c = i % tmp.colNum;
255         for (int j = 0; j < m1.colNum; j++)
256         {
257             tmp.data[i] = tmp.data[i] + m1.data[r * m1.colNum + j] * m2.data[j * m2.colNum
+ c];
258         }
259     }
260     return tmp;
261 }
262
263 void testMatrix()
264 {
265     //整数类的模板实例。
266     //加减, 索引, 乘法
267     int a[6] = {1, 1, 1, 1, 1, 1};
268     Matrix<int> m1(2, 3, a), m2(3, 2, a);
269     m1.print();
270     m2.print();
271     (m1 + m1).print();
272     (m1 - m1).print();
273     int b[6] = {1, 2, 3, 4, 5, 6};
274     Matrix<int> m3(2, 3, b);
275     for (int i = 0; i < 2; i++)
276     {
277         for (int j = 0; j < 3; j++)
278         {
279             cout << m3[i][j] << endl;
280         }
281     }
282     (m1 * m2).print();

```

```

283     (m2 * m1).print();
284
285     Complex c[6] = {Complex(1, 1), Complex(1, 1), Complex(1, 1), Complex(1, 1), Complex(1,
1), Complex(1, 1)};
286     Matrix<Complex> m4(2, 3, c), m5(3, 2, c);
287     m4.print();
288     m5.print();
289     (m4 + m4).print();
290     (m4 - m4).print();
291     for (int i = 0; i < 2; i++)
292     {
293         for (int j = 0; j < 3; j++)
294         {
295             cout << m4[i][j] << endl;
296         }
297     }
298     (m4 * m5).print();
299     (m5 * m4).print();
300
301     return;
302 }
303 int main()
304 {
305     testComplex();
306     testMatrix();
307 }

```

运行结果

```

1  1+1j
2  2+2j
3  0
4  a<b:1
5  0+4j
6  0.5
7  3+3j
8  0
9  1  1  1
10 1  1  1
11 1  1
12 1  1
13 1  1
14 2  2  2
15 2  2  2
16 0  0  0
17 0  0  0
18 1
19 2
20 3
21 4
22 5
23 6
24 3  3
25 3  3
26 2  2  2
27 2  2  2
28 2  2  2
29 1+1i 1+1i 1+1i
30 1+1i 1+1i 1+1i
31 1+1i 1+1i

```

```
32 1+1i 1+1i
33 1+1i 1+1i
34 2+2i 2+2i 2+2i
35 2+2i 2+2i 2+2i
36 0+0i 0+0i 0+0i
37 0+0i 0+0i 0+0i
38 1+1i
39 1+1i
40 1+1i
41 1+1i
42 1+1i
43 1+1i
44 1+1i 1+1i 1+1i
45 1+1i 1+1i 1+1i
46 1+1i 1+1i
47 1+1i 1+1i
48 1+1i 1+1i
49 2+2i 2+2i 2+2i
50 2+2i 2+2i 2+2i
51 0+0i 0+0i 0+0i
52 0+0i 0+0i 0+0i
53 1+1i
54 1+1i
55 1+1i
56 1+1i
57 1+1i
58 1+1i
59 0+6i 0+6i
60 0+6i 0+6i
61 0+4i 0+4i 0+4i
62 0+4i 0+4i 0+4i
63 0+4i 0+4i 0+4i
```

与python复数矩阵的计算进行对比，验证是正确的。