Assignment 9

191300051[1]

Wang Fuyun[1]

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

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

//实现能够有多种数据类型的最大堆。
template <class Type>
class MaxHeap
{
private:
    Type *data;
    int size;     //当前大小
    int capacity; //总容量
    int parent(int i)
    {
        //-1/2=-1;
        return (i - 1) / 2;
    }
    int left(int i)
    {
        return i * 2 + 1;
    }
    int right(int i)
    {
        return (i + 1) * 2;
    }
    void heapify(int i)
    {
        int l = left(i);
        int r = right(i);
        int largest = i;
        if (l < size && data[l] > data[largest])
```

```
        {
            largest = l;
        }
        if (r < size && data[r] > data[largest])
        {
            largest = r;
        }
        if (largest != i)
        {
            swap(data[i], data[largest]);
            heapify(largest);
        }
    }

public:
    MaxHeap(int c = 10) : data(new Type[c]), size(0), capacity(c)
    {
    }
    //直接从数组进行建堆
    //则指针A指向的数组不可变。
    MaxHeap(const Type A[], int n, int c) : data(new Type[c]), size(n), capacity(c)
    {
        for (int i = 0; i < n; i++)
        {
            data[i] = A[i];
        }
        for (int i = (size - 1) / 2; i >= 0; i--)
        {
            heapify(i);
        }
    }
    //默认的析构函数不会删除掉动态变量。
    ~MaxHeap()
    {
        delete data;
    }
    bool Insert(Type element) //插入一个元素
    {
        data[size] = element;
        int i = size;
        size++;
        while (i >= 0 && data[parent(i)] < data[i])
        {
            swap(data[parent(i)], data[i]);
            i = parent(i);
        }
    }
    //让出0位置的元素，将末尾元素置于0位置，并执行heapify。
    Type DeleteMax() //找出最大的元素返回，并进行删除
    {
        Type ans = data[0];
        size--;
        data[0] = data[size];
        heapify(0);
        return ans;
    }
    bool IsFull() //是否为满
    {
        return size == capacity;
    }
    bool IsEmpty() //是否为空
```

```
    {
        return size == 0;
    }
    void Print() //打印
    {
        cout << "PRINT THE ELEMENT IN HEAP\n";
        for (int i = 0; i < size; i++)
        {
            cout << data[i] << " ";
            if (left(i) < size)
            {
                cout << data[left(i)]<<" ";
            }
            if (right(i)<size)
            {
                cout<<data[right(i)]<<" ";
            }
            cout<<endl;
        }
    }
};

int main()
{
    cout << "TEST1\n";
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    MaxHeap<int> h1(a, 10, 20);
    h1.Print();
    h1.Insert(11);
    h1.Print();
    h1.Insert(100);
    h1.Insert(1000);
    cout << h1.DeleteMax() << endl;
    h1.Print();

    cout << "TEST2\n";
    MaxHeap<int> h2(20);
    for (int i = 1; i < 11; i++)
    {
        h2.Insert(i);
    }
    h2.Print();
    h2.Insert(11);
    h2.Print();
    h2.Insert(100);
    h2.Insert(1000);
    cout << h2.DeleteMax() << endl;
    h2.Print();
}
```

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。

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

```
5 2
6
3
1
4
2
PRINT THE ELEMENT IN HEAP
11 10 7
10 8 9
7 6 3
8 1 4
9 2 5
6
3
1
4
2
5
1000
PRINT THE ELEMENT IN HEAP
100 10 11
10 8 9
11 7 3
8 1 4
9 2 5
7 6
3
1
4
2
5
6
TEST2
PRINT THE ELEMENT IN HEAP
10 9 6
9 7 8
6 2 5
7 1 4
8 3
2
5
1
4
3
PRINT THE ELEMENT IN HEAP
11 10 6
10 7 9
6 2 5
7 1 4
9 3 8
2
5
1
4
3
100 10 11
10 7 9
11 6 5
7 1 4
9 3 8
6 2
```

```
5
1
4
3
8
2
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

```cpp
//实现支持多种数据类型，包括自建的复数类的矩阵类
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cassert>
using namespace std;

//在Complex类中，可以直接访问其他Complex对象的private资源。
class Complex
{
private:
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i)
    {
    }
    void print()
    {
        if (imag == 0)
        {
            cout << real << endl;
        }
        else if (imag > 0)
        {
            cout << real << "+" << imag << "j" << endl;
        }
        else
        {
            cout << real << imag << "j" << endl;
        }
    }
    double modulus() const
    {
        return real * real + imag * imag;
    }
    //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
    //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
    //但是如果没有指针接受它的话则内存泄漏
    //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。
    //想要在左边就能用，必须写成类内的重载函数了。
```

```
    Complex operator-() const
    {
        Complex tmp;
        tmp.real = -real;
        tmp.imag = -imag;
        return tmp;
    }
    //=号的重载也必须是成员函数，与拷贝构造函数基本一致。
    Complex &operator=(const Complex &c)
    {
        real = c.real;
        imag = c.imag;
        return *this;
    }
    //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。
    friend bool operator==(const Complex &c1, const Complex &c2);
    friend bool operator!=(const Complex &c1, const Complex &c2);
    friend bool operator>(const Complex &c1, const Complex &c2);
    friend bool operator>=(const Complex &c1, const Complex &c2);
    friend bool operator<(const Complex &c1, const Complex &c2);
    friend bool operator<=(const Complex &c1, const Complex &c2);
    friend Complex operator+(const Complex &c1, const Complex &c2);
    friend Complex operator-(const Complex &c1, const Complex &c2);
    friend Complex operator*(const Complex &c1, const Complex &c2);
    friend Complex operator/(const Complex &c1, const Complex &c2);
    friend ostream &operator<<(ostream &output, const Complex &c);
};
bool operator==(const Complex &c1, const Complex &c2)
{
    return c1.real == c2.real && c1.imag == c2.imag;
}
bool operator!=(const Complex &c1, const Complex &c2)
{
    return !(c1 == c2);
}
bool operator>(const Complex &c1, const Complex &c2)
{
    return c1.modulus() > c2.modulus();
}
bool operator>=(const Complex &c1, const Complex &c2)
{
    return c1.modulus() >= c2.modulus();
}
bool operator<(const Complex &c1, const Complex &c2)
{
    return c1.modulus() < c2.modulus();
}
bool operator<=(const Complex &c1, const Complex &c2)
{
    return c1.modulus() <= c2.modulus();
}
Complex operator+(const Complex &c1, const Complex &c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
Complex operator-(const Complex &c1, const Complex &c2)
{
    return Complex(c1.real - c2.real, c1.imag - c2.imag);
}
Complex operator*(const Complex &c1, const Complex &c2)
{
```

```
    return Complex(c1.real * c2.real - c1.imag * c2.imag, c1.real * c2.imag + c1.imag * c2.real);
}
Complex operator/(const Complex &c1, const Complex &c2)
{
    double d = c2.modulus();
    if (d == 0)
    {
        cerr << "Error in operation / of Complex" << endl;
        exit(-1);
    }
    else
    {
        return Complex((c1.real * c2.real + c1.imag * c2.imag) / d, (c1.imag * c2.real - c1.real * c2.imag) / d);
    }
}
ostream &operator<<(ostream &output, const Complex &c)
{
    int num = c.imag;

    if (num >= 0)
    {
        output << c.real << "+" << c.imag << "i";
    }
    else
    {
        output << c.real << c.imag << "i";
    }

    return output;
}

void testComplex()
{
    Complex a(1, 1), b(2, 2), c(0, 0);
    a.print();
    b.print();
    c.print();
    cout << "a<b:" << (a < b) << endl;
    (a * b).print();
    (a / b).print();
    (a + b).print();
    (a * c).print();

    //divided by zero exit(-1);
    // (a / c).print();
}

template <class T>
class Matrix
{
private:
    int colNum;
    int rowNum;
    T *data;

public:
    Matrix(int r = 0, int c = 0)
    {
        colNum = c;
```

```
        rowNum = r;
        data = new T[colNum * rowNum];
        // for (int i=0;i<r*c;i++)
        // {
        //
        // }
        memset(data, 0, sizeof(T) * c * r);
    }
    Matrix(int r, int c, T a[])
    {
        colNum = c;
        rowNum = r;
        data = new T[colNum * rowNum];
        for (int i = 0; i < r * c; i++)
        {
            data[i] = a[i];
        }
    }
    void print()
    {
        for (int i = 0; i < rowNum; i++)
        {
            for (int j = 0; j < colNum; j++)
            {
                cout << data[i * colNum + j] << "  ";
            }
            cout << endl;
        }
    }
    //赋值操作
    Matrix &operator=(const Matrix &m)
    {
        colNum = m.colNum;
        rowNum = m.rowNum;
        delete data;
        data = new T[colNum * rowNum];
        memset(data, m.data, sizeof(T) * colNum * rowNum);
        return *this;
    }

    //实现矩阵类
    Matrix operator-()
    {
        Matrix m(rowNum, colNum);
        for (int i = 0; i < colNum * rowNum; i++)
        {
            m[i] = -data[i];
        }
    }
    T *operator[](int index)
    {
        return &data[index * colNum];
    }
    //函数模板的声明上面也要加!
    //友元函数的class和模板类的class还不能一样?
    template <class Ty>
    friend Matrix<Ty> operator+(const Matrix<Ty> &m1, const Matrix<Ty> &m2);
    template <class Ty>
    friend Matrix<Ty> operator-(const Matrix<Ty> &m1, const Matrix<Ty> &m2);
    template <class Ty>
    friend Matrix<Ty> operator*(const Matrix<Ty> &m1, const Matrix<Ty> &m2);
```

```
};
template <class T>
Matrix<T> operator+(const Matrix<T> &m1, const Matrix<T> &m2)
{
    assert(m1.colNum == m2.colNum && m1.rowNum == m2.rowNum);
    Matrix<T> tmp(m1.rowNum, m1.colNum);
    for (int i = 0; i < tmp.colNum * tmp.rowNum; i++)
    {
        tmp.data[i] = m1.data[i] + m2.data[i];
    }
    return tmp;
}
template <class T>
Matrix<T> operator-(const Matrix<T> &m1, const Matrix<T> &m2)
{
    assert(m1.colNum == m2.colNum && m1.rowNum == m2.rowNum);
    Matrix<T> tmp(m1.rowNum, m1.colNum);
    for (int i = 0; i < tmp.colNum * tmp.rowNum; i++)
    {
        tmp.data[i] = m1.data[i] - m2.data[i];
    }
    return tmp;
}
template <class T>
Matrix<T> operator*(const Matrix<T> &m1, const Matrix<T> &m2)
{
    assert(m1.colNum == m2.rowNum);
    Matrix<T> tmp(m1.rowNum, m2.colNum);
    for (int i = 0; i < tmp.rowNum * tmp.colNum; i++)
    {
        int r = i / tmp.colNum;
        int c = i % tmp.colNum;
        for (int j = 0; j < m1.colNum; j++)
        {
            tmp.data[i] = tmp.data[i] + m1.data[r * m1.colNum + j] * m2.data[j * m2.colNum
+ c];
        }
    }
    return tmp;
}

void testMatrix()
{
    //整数类的模板实例。
    //加减，索引，乘法
    int a[6] = {1, 1, 1, 1, 1, 1};
    Matrix<int> m1(2, 3, a), m2(3, 2, a);
    m1.print();
    m2.print();
    (m1 + m1).print();
    (m1 - m1).print();
    int b[6] = {1, 2, 3, 4, 5, 6};
    Matrix<int> m3(2, 3, b);
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << m3[i][j] << endl;
        }
    }
    (m1 * m2).print();
```

```
    (m2 * m1).print();

    Complex c[6] = {Complex(1, 1), Complex(1, 1), Complex(1, 1), Complex(1, 1), Complex(1, 1), Complex(1, 1)};
    Matrix<Complex> m4(2, 3, c), m5(3, 2, c);
    m4.print();
    m5.print();
    (m4 + m4).print();
    (m4 - m4).print();
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << m4[i][j] << endl;
        }
    }
    (m4 * m5).print();
    (m5 * m4).print();

    return;
}
int main()
{
    testComplex();
    testMatrix();
}
```

运行结果

```
1+1j
2+2j
0
a<b:1
0+4j
0.5
3+3j
0
1  1  1
1  1  1
1  1
1  1
1  1
2  2  2
2  2  2
0  0  0
0  0  0
1
2
3
4
5
6
3  3
3  3
2  2  2
2  2  2
2  2  2
1+1i  1+1i  1+1i
1+1i  1+1i  1+1i
1+1i  1+1i
```

```
1+1i  1+1i
1+1i  1+1i
2+2i  2+2i  2+2i
2+2i  2+2i  2+2i
0+0i  0+0i  0+0i
0+0i  0+0i  0+0i
1+1i
1+1i
1+1i
1+1i
1+1i
1+1i
1+1i  1+1i  1+1i
1+1i  1+1i  1+1i
1+1i  1+1i
1+1i  1+1i
1+1i  1+1i
2+2i  2+2i  2+2i
2+2i  2+2i  2+2i
0+0i  0+0i  0+0i
0+0i  0+0i  0+0i
1+1i
1+1i
1+1i
1+1i
1+1i
1+1i
0+6i  0+6i
0+6i  0+6i
0+4i  0+4i  0+4i
0+4i  0+4i  0+4i
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。