Solution9

191220008 陈南曈

概念题

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

错误。

函数模板的实例化通常是隐式的：

- 由编译程序根据函数调用的实参类型自动地把函数模板实例化为具体的函数。
- 这种确定函数模板实例的过程叫做模板实参推导（template argument deduction）

（2）函数模板的实例化由编译器实现

错误。

有时，编译程序无法根据调用时的实参类型来确定所调用的模板实例函数，这时，需要在程序中显式地实例化函数模板。

（3）一个类定义中，只要有一个函数模板，则这个类是类模板

正确。

（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

错误。

首先，类模板的成员函数不一定是函数模板，还可能是普通的成员函数。

其次，类模板实例化后，成员函数不会随之实例化，而是在调用该成员函数（且能看见函数源码）时，进行实例化。

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中的静态成员。只有相同类模板示例才会共享同一个静态成员。

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?

使用一个模板之前首先要对其实例化（用一个具体的类型去替代模板的类型参数），而实例化是在编译时刻进行的，它一定要见到相应的源代码，否则无法实例化。

解决上述问题的通常做法是把模板的定义和实现都放在头文件中。使用者通过包含这个头文件，把模板的源代码全包含进来，以备实例化所需。

编程题

1、用模板类实现一个最大堆。

```
template<class Type>
class MaxHeap {
private:
    Type* Data;
    int Size; //当前大小
    int Capacity; //总容量
    void SiftDown(int start, int m); //从 start 到 m 下滑调整成为最大堆
    void SiftUp(int start); //从 start 到 0 上滑调整成为最大堆
public:
    MaxHeap(); //默认构造函数，容量为10
    MaxHeap(int Capacity);
    ~MaxHeap();
    bool Insert(Type element); //插入一个元素
    Type DeleteMax(); //找出最大的元素返回，并进行删除
    bool IsFull(); //是否为满
    bool IsEmpty(); //是否为空
    void Print(); //打印
};

template<class Type>
void MaxHeap <Type>::SiftDown(int start, int m)
{
    int i = start, j = 2 * i + 1;
    Type temp = Data[i];
    while (j <= m)
    {
        if (j < m && Data[j] < Data[j + 1])
            j++;
        if (temp >= Data[j])
            break;
        else
        {
            Data[i] = Data[j];
            i = j;
            j = 2 * j + 1;
        }
    }
    Data[i] = temp;
}

template<class Type>
void MaxHeap <Type>::SiftUp(int start)
{
    int j = start, i = (j - 1) / 2;
    Type temp = Data[j];
    while (j > 0)
    {
        if (Data[i] >= temp)
            break;
        else
        {
            Data[j] = Data[i];
            j = i;
            i = (i - 1) / 2;
        }
    }
    Data[j] = temp;
}

template<class Type>
MaxHeap <Type>::MaxHeap()
{
    Capacity = 10;
    Data = new Type[Capacity];
    Size = 0;
}

template<class Type>
MaxHeap <Type>::MaxHeap(int Capacity)
{
    this->Capacity = Capacity;
    Data = new Type[Capacity];
    Size = 0;
}

template<class Type>
MaxHeap <Type>::~MaxHeap()
{
    Capacity = 0;
    delete[]Data;
    Data = NULL;
    Size = 0;
}

template<class Type>
bool MaxHeap <Type>::Insert(Type element)
{
    if(IsFull())
    {
        cout << "Heap Full" << endl;
        return false;
    }
    Data[Size] = element;
    SiftUp(Size);
    Size++;
    return true;
}

template<class Type>
Type MaxHeap <Type>::DeleteMax()
{
    if(IsEmpty())
    {
        cout << "Heap Empty" << endl;
        return false;
    }
    Type element;
    element = Data[0];
    Data[0] = Data[Size - 1];
    Size--;
    SiftDown(0, Size - 1);
    return true;
}

template<class Type>
bool MaxHeap <Type>::IsFull()
{
    return Size == Capacity;
}

template<class Type>
bool MaxHeap <Type>::IsEmpty()
{
    return Size == 0;
}

template<class Type>
void MaxHeap <Type>::Print()
{
    for (int i = 0; i < Size; i++)
        cout << Data[i] << " ";
    cout << endl;
}
```

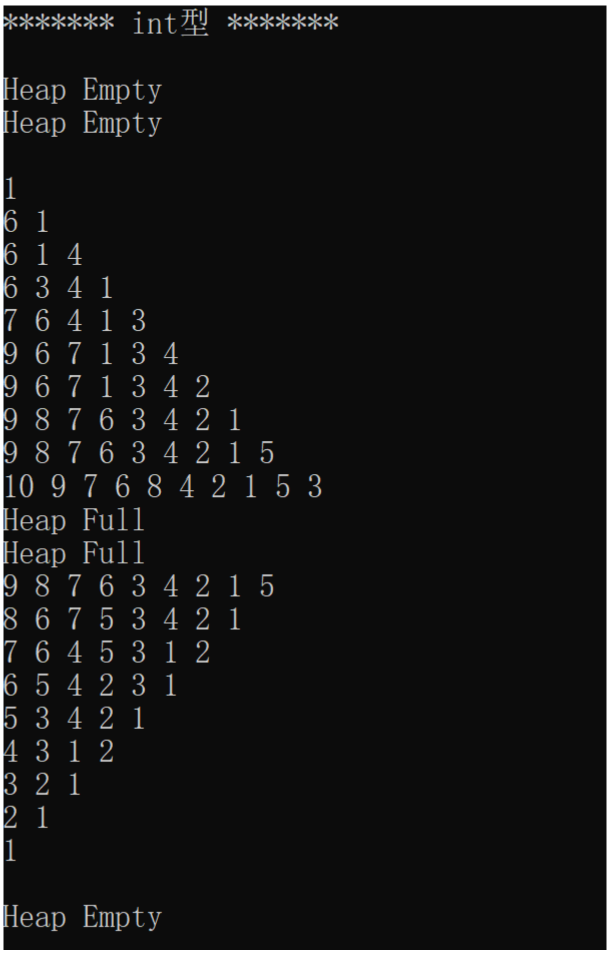
测试效果：

（1）int型

测试代码：

```
int main()
{
    cout << "******* int型 *******" << endl << endl;
    MaxHeap<int> heap1;
    if (heap1.IsEmpty())
        cout << "Heap Empty" << endl;
    heap1.DeleteMax();
    heap1.Print();
    heap1.Insert(1);
    heap1.Print();
    if (heap1.IsEmpty())
        cout << "Heap Empty" << endl;
    heap1.Insert(6);
    heap1.Print();
    heap1.Insert(4);
    heap1.Print();
    heap1.Insert(3);
    heap1.Print();
    heap1.Insert(7);
    heap1.Print();
    heap1.Insert(9);
    heap1.Print();
    heap1.Insert(2);
    heap1.Print();
    heap1.Insert(8);
    heap1.Print();
    heap1.Insert(5);
    heap1.Print();
    if (heap1.IsFull())
        cout << "Heap Full" << endl;
    heap1.Insert(10);
    heap1.Print();
    if (heap1.IsFull())
        cout << "Heap Full" << endl;
    heap1.Insert(10);
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();
    heap1.Print();
    heap1.DeleteMax();

    return 0;
}
```

测试效果:



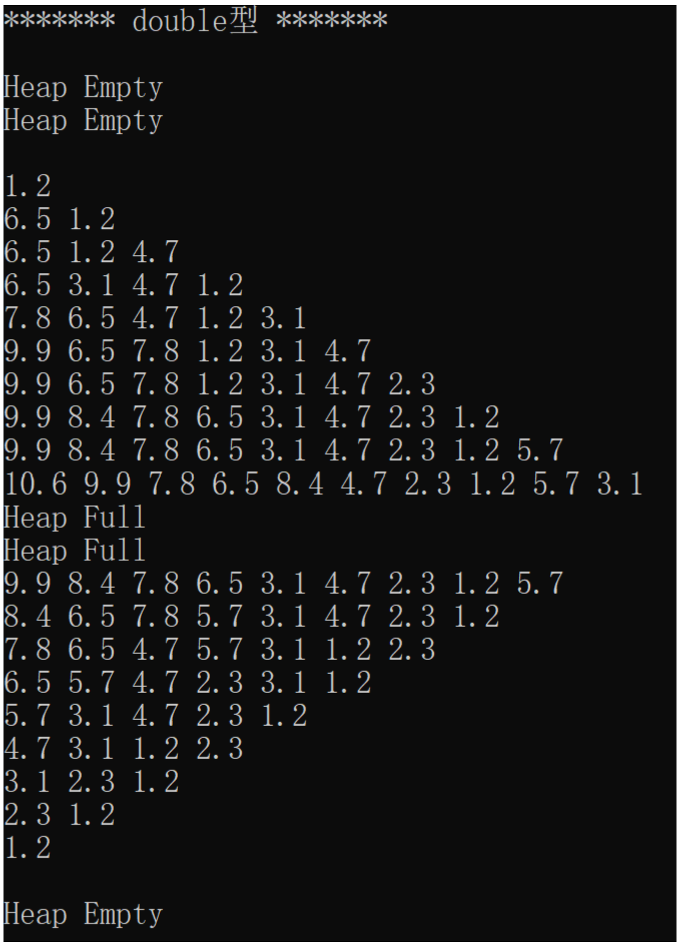
```
******* int型 *******

Heap Empty
Heap Empty

1
6 1
6 1 4
6 3 4 1
7 6 4 1 3
9 6 7 1 3 4
9 6 7 1 3 4 2
9 8 7 6 3 4 2 1
9 8 7 6 3 4 2 1 5
10 9 7 6 8 4 2 1 5 3
Heap Full
Heap Full
9 8 7 6 3 4 2 1 5
8 6 7 5 3 4 2 1
7 6 4 5 3 1 2
6 5 4 2 3 1
5 3 4 2 1
4 3 1 2
3 2 1
2 1
1

Heap Empty
```

（2）double型

测试代码：

```cpp
int main()
{
    cout << "******* double型 *******" << endl << endl;
    MaxHeap<double> heap2;
    if (heap2.IsEmpty())
        cout << "Heap Empty" << endl;
    heap2.DeleteMax();
    heap2.Print();
    heap2.Insert(1.2);
    heap2.Print();
    if (heap2.IsEmpty())
        cout << "Heap Empty" << endl;
    heap2.Insert(6.5);
    heap2.Print();
    heap2.Insert(4.7);
    heap2.Print();
    heap2.Insert(3.1);
    heap2.Print();
    heap2.Insert(7.8);
    heap2.Print();
    heap2.Insert(9.9);
    heap2.Print();
    heap2.Insert(2.3);
    heap2.Print();
    heap2.Insert(8.4);
    heap2.Print();
    heap2.Insert(5.7);
    heap2.Print();
    if (heap2.IsFull())
        cout << "Heap Full" << endl;
    heap2.Insert(10.6);
    heap2.Print();
    if (heap2.IsFull())
        cout << "Heap Full" << endl;
    heap2.Insert(10.2);
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    heap2.Print();
    heap2.DeleteMax();
    return 0;
}
```

测试效果：



```
******* double型 *******

Heap Empty
Heap Empty

1.2
6.5 1.2
6.5 1.2 4.7
6.5 3.1 4.7 1.2
7.8 6.5 4.7 1.2 3.1
9.9 6.5 7.8 1.2 3.1 4.7
9.9 6.5 7.8 1.2 3.1 4.7 2.3
9.9 8.4 7.8 6.5 3.1 4.7 2.3 1.2
9.9 8.4 7.8 6.5 3.1 4.7 2.3 1.2 5.7
10.6 9.9 7.8 6.5 8.4 4.7 2.3 1.2 5.7 3.1
Heap Full
Heap Full
9.9 8.4 7.8 6.5 3.1 4.7 2.3 1.2 5.7
8.4 6.5 7.8 5.7 3.1 4.7 2.3 1.2
7.8 6.5 4.7 5.7 3.1 1.2 2.3
6.5 5.7 4.7 2.3 3.1 1.2
5.7 3.1 4.7 2.3 1.2
4.7 3.1 1.2 2.3
3.1 2.3 1.2
2.3 1.2
1.2

Heap Empty
```

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

（1）int;

（2）复数类（你得首先自己实现这个类Complex，请复习复数乘法和加法）。

注意：请结合最近的课程知识来完成编程题，自行构造测试用例来测试实现的功能，并提供运行截图。

实现代码:

```
class Complex
{
private:
    double real;
    double image;
public:
    Complex(double _real = 0.0, double _image = 0.0) :real{ _real }, image{
_image } {}
    Complex(const Complex& complex) :real{ complex.real }, image{ complex.image
}{}
    bool operator == (const Complex& complex) {
        return real == complex.real && image == complex.image;
    }
    bool operator != (const Complex& complex) {
        return real != complex.real || image != complex.image;
    }
```

```cpp
    Complex operator + (const Complex& complex) {
        return Complex(real + complex.real, image + complex.image);
    }

    Complex operator - (const Complex& complex) {
        return Complex(real - complex.real, image - complex.image);
    }

    Complex operator * (const Complex& complex) {
        double _real = real * complex.real - image * complex.image;
        double _image = image * complex.real + real * complex.image;
        return Complex(_real, _image);
    }

    Complex operator / (const Complex& complex) {
        double _real = (real * complex.real + image * complex.image) /
(complex.real * complex.real + complex.image * complex.image);
        double _image = (image * complex.real - real * complex.image) /
(complex.real * complex.real + complex.image * complex.image);
        return Complex(_real, _image);
    }
    Complex& operator = (const Complex& complex)
    {
        if (this != &complex)
        {
            real = complex.real;
            image = complex.image;
        }
        return *this;
    }
    Complex& operator += (const Complex& complex) {
        real += complex.real;
        image += complex.image;
        return *this;
    }
    friend istream& operator >> (istream& in, Complex& complex);
    friend ostream& operator << (ostream& out, Complex& complex);
};

istream& operator >> (istream& in, Complex& complex) {
    in >> complex.real >> complex.image;
    return in;
}

ostream& operator << (ostream& out, Complex& complex)
{
    out << "(" << complex.real;
    if (complex.image >= 0)
    {
        out << "+" << complex.image << "i)";
    }
    else
    {
        out << complex.image << "i)";
    }
    return out;
}
```

```
template <class Type>
class Matrix
{
private:
    Type** p_data; //表示矩阵数据
    int row, col; //表示矩阵的行数和列数
public:
    Matrix(int r, int c); //构造函数
    Matrix(const Matrix <Type>& m); //拷贝构造函数
    ~Matrix(); //析构函数
    Type*& operator[] (int i); //重载[], 对于Matrix对象m, 能够通过m[i][j]访问第i+1行、
第j+1列元素
    Matrix<Type>& operator = (const Matrix<Type>& m); //重载=, 实现矩阵整体赋值, 若
行/列不等, 归还空间并重新分配
    bool operator == (const Matrix<Type>& m) const; //重载==, 判断矩阵是否相等
    Matrix<Type> operator + (const Matrix<Type>& m) const; //重载+, 完成矩阵加法, 可
假设两矩阵满足加法条件(两矩阵行、列分别相等)
    Matrix<Type> operator * (const Matrix<Type>& m) const; //重载*, 完成矩阵乘法, 可
假设两矩阵满足乘法条件(this.col = m.row)
    void print();
};

template <class Type>
Matrix<Type>::Matrix(int r, int c)
{
    row = r;
    col = c;
    p_data = new Type* [row];
    for (int i = 0; i < row; i++)
    {
        p_data[i] = new Type[col];
        for (int j = 0; j < col; j++)
            p_data[i][j] = 0;
    }
}

template <class Type>
Matrix<Type>::Matrix(const Matrix<Type>& m)
{
    row = m.row;
    col = m.col;
    p_data = new Type* [row];
    for (int i = 0; i < row; i++)
    {
        p_data[i] = new Type[col];
        for (int j = 0; j < col; j++)
        {
            p_data[i][j] = m.p_data[i][j];
        }
    }
}

template <class Type>
Matrix<Type>::~Matrix()
{
    for (int i = 0; i < row; i++)
        delete[]p_data[i];
    delete[]p_data;
```

```cpp
}

template <class Type>
Type*& Matrix<Type>::operator[] (int i)
{
    return p_data[i];
}

template <class Type>
Matrix <Type>& Matrix<Type>::operator = (const Matrix<Type>& m)
{
    if (row != m.row || col != m.col)
    {
        this->~Matrix();
        p_data = new Type* [m.row];
        for (int i = 0; i < m.row; i++)
        {
            p_data[i] = new Type[m.col];
            for (int j = 0; j < m.col; j++)
                p_data[i][j] = 0;
        }
        row = m.row;
        col = m.col;
    }
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            p_data[i][j] = m.p_data[i][j];
        }
    }
    return *this;
}

template <class Type>
bool Matrix<Type>::operator == (const Matrix<Type>& m) const
{
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            if (p_data[i][j] != m.p_data[i][j])
                return false;
        }
    }
    return true;
}

template <class Type>
Matrix<Type> Matrix<Type>::operator + (const Matrix<Type>& m) const
{
    Matrix<Type> matrix(row, col);
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            matrix.p_data[i][j] = p_data[i][j] + m.p_data[i][j];
        }
    }
    return matrix;
}

template <class Type>
Matrix<Type> Matrix<Type>::operator * (const Matrix<Type>& m) const
{
    Matrix<Type> matrix(row, m.col);
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < m.col; j++)
        {
            for (int k = 0; k < col; k++)
            {
                matrix.p_data[i][j] += p_data[i][k] * m.p_data[k][j];
            }
        }
    }
    return matrix;
}

template <class Type>
void Matrix<Type>::print()
{
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            cout << p_data[i][j] << ' ';
        }
        cout << endl;
    }
}
```

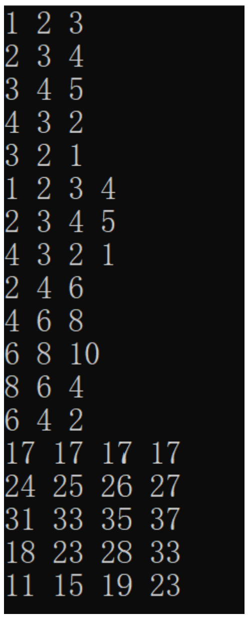
注：为测试方便，在测试时暂时将矩阵中的成员变量改为public访问。

（1）int型

测试代码：

```
int main()
{
    Matrix<int> m1(5, 3), m2(3, 4);
    for (int i = 0; i < m1.row; i++)
        for (int j = 0; j < m1.col; j++)
            cin >> m1.p_data[i][j];
    for (int i = 0; i < m2.row; i++)
        for (int j = 0; j < m2.col; j++)
            cin >> m2.p_data[i][j];
    Matrix<int> m3(5, 3);
    m3 = m1 + m1;
    m3.print();
    Matrix<int> m4(5, 4);
    m4 = m1 * m2;
    m4.print();
    return 0;
}
```

测试效果：



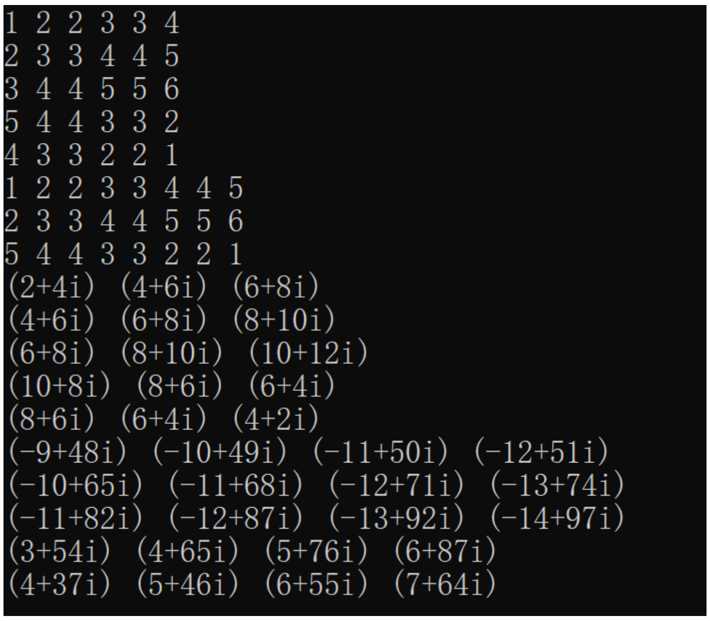
```
1 2 3
2 3 4
3 4 5
4 3 2
3 2 1
1 2 3 4
2 3 4 5
4 3 2 1
2 4 6
4 6 8
6 8 10
8 6 4
6 4 2
17 17 17 17
24 25 26 27
31 33 35 37
18 23 28 33
11 15 19 23
```

（2）Complex类

测试代码：

```
int main()
{
    Matrix<Complex> m1(5, 3), m2(3, 4);
    for (int i = 0; i < m1.row; i++)
        for (int j = 0; j < m1.col; j++)
            cin >> m1.p_data[i][j];
    for (int i = 0; i < m2.row; i++)
        for (int j = 0; j < m2.col; j++)
            cin >> m2.p_data[i][j];
    Matrix<Complex> m3(5, 3);
    m3 = m1 + m1;
    m3.print();
    Matrix<Complex> m4(5, 4);
    m4 = m1 * m2;
    m4.print();
    return 0;
}
```

测试效果：



```
1 2 2 3 3 4
2 3 3 4 4 5
3 4 4 5 5 6
5 4 4 3 3 2
4 3 3 2 2 1
1 2 2 3 3 4 4 5
2 3 3 4 4 5 5 6
5 4 4 3 3 2 2 1
(2+4i) (4+6i) (6+8i)
(4+6i) (6+8i) (8+10i)
(6+8i) (8+10i) (10+12i)
(10+8i) (8+6i) (6+4i)
(8+6i) (6+4i) (4+2i)
(-9+48i) (-10+49i) (-11+50i) (-12+51i)
(-10+65i) (-11+68i) (-12+71i) (-13+74i)
(-11+82i) (-12+87i) (-13+92i) (-14+97i)
(3+54i) (4+65i) (5+76i) (6+87i)
(4+37i) (5+46i) (6+55i) (7+64i)
```