Assignment 9 答案

一、概念简答题

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
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

解答：

1. 错误，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，函数模板和类模板的实例化都是由编译器实现的；

3. 错误，类模板中的成员函数都是函数模板；

4. 错误，类模板实例化后，没有用到的成员函数没有实例化；

2. 在类模版中使用静态成员有什么要注意的地方？

类模板中的静态成员仅属于实例化后的类，不同类模板实例之间不共享模板中的静态成员

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，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。

二、代码编程题

1. 用模板类实现一个最大堆

代码示例：

```
 template <class T>
class MaxHeap {
private:
    static const int INIT_CAP = 10;
    T* items;
    int size;
    int cap;
    static int parent(int n) { //父节点
        return n / 2;
    }
    static int left(int n) { //左子结点
        return n * 2;
    }
    static int right(int n) { //右子结点
        return n * 2 + 1;
    }
    void swap(int a, int b) {
        T temp = items[a];
        items[a] = items[b];
        items[b] = temp;
    }
    void swim(int n) { //上滑
        if (n == 1) {
            return;
        }
        if (items[n] > items[parent(n)]) {
            swap(parent(n), n);
            swim(parent(n));
        }
    }
    void sink(int n) { //下沉
        if (left(n) > size) {
            return;
        } else if (right(n) > size) {
            if (items[left(n)] > items[n]) {
                swap(n, left(n));
            }
        } else {
            int bestIndex = items[left(n)] > items[right(n)] ? left(n) :
right(n);
            if (items[bestIndex] > items[n]) {
                swap(n, bestIndex);
                sink(bestIndex);
            }
        }
    }
public:
    MaxHeap() {
        items = new T[INIT_CAP + 1]; //下标从1开始
        size = 0;
        cap = INIT_CAP;
    }
    MaxHeap(int capacity) {
        items = new T[capacity + 1];
        size = 0;
        cap = capacity;
    }
    ~MaxHeap() {
        delete[] items;
    }
    bool isFull() const {
        return size == cap;
    }
    bool isEmpty() const {
        return size == 0;
    }
    void print() const {
        for (int i = 1; i <= size; ++i) {
            std::cout << items[i] << ' ';
        }
        std::cout << std::endl;
    }
    bool insert(T element) {
        if (isFull()) {
            return false;
        }
        items[++size] = element;
        swim(size);
        return true;
    }
    T deleteMax() {
        T element = items[1];
        items[1] = items[size--];
        sink(1);
        return element;
    }
};
```

测试用例：

```
int main() {
    MaxHeap<int> intHeap;
    MaxHeap<double> doubleHeap(11);
    std::default_random_engine e(1);
    std::uniform_int_distribution<int> i(-100, 100);
    std::uniform_real_distribution<double> r(-100.0, 100.0);
    while (!doubleHeap.isFull()) {
        int n = i(e);
        double x = r(e);
        if (!intHeap.insert(n)) {
            std::cout << "IntHeap Full!" << std::endl;
        }
        if (!doubleHeap.insert(x)) {
            std::cout << "doubleHeap Full!" << std::endl;
        }
    }
    intHeap.print();
    doubleHeap.print();
    while (!intHeap.isEmpty()) {
        std::cout << intHeap.deleteMax() << ' ';
    }
    std::cout << std::endl;
    while (!doubleHeap.isEmpty()) {
        std::cout << doubleHeap.deleteMax() << ' ';
    }
    std::cout << std::endl;
    return 0;
}
```

输出结果：

```
IntHeap Full!
87 70 6 67 31 -91 -23 -100 38 -8
86.0873 51.1211 35.8593 3.88327 40.2381 -98.4604 -16.5028 -56.2082 -81.607
-89.3077 -47.5094
87 70 67 38 31 6 -8 -23 -91 -100
86.0873 51.1211 40.2381 35.8593 3.88327 -16.5028 -47.5094 -56.2082 -81.607
-89.3077 -98.4604
```

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。**

代码示例：

```
class Complex {
    int real;
    int image;
public:
    Complex() { real = 0; image = 0; }
    Complex(int r, int i) { real = r; image = i; }
    bool operator == (Complex t) const{
        return real == t.real && image == t.image;
    }
    bool operator != (Complex t) const{
        return !((*this) == t);
    }
    Complex operator +(const Complex t) const{
        Complex n(0, 0);
        n.real = real + t.real;
        n.image = image + t.image;
        return n;
    }
    Complex operator *(const Complex t) const{
        Complex n(0, 0);
        n.real = real * t.real - image * t.image;
        n.image = image * t.real + real * t.image;
        return n;
    }

    friend istream& operator >> (istream& input, Complex& C) {
        string s;
        input >> s;
        int pos = s.find("+", 0);
        string sTmp = s.substr(0, pos);
        C.real = atoi(sTmp.c_str());
        sTmp = s.substr(pos + 1, s.length() - pos - 1);
        C.image = atoi(sTmp.c_str());
        return input;
    }
```

```cpp
    friend ostream& operator << (ostream& output, const Complex& C) {
        output << C.real << "+" << C.image << "i";
        return output;
    }
};

template <class T>
class Matrix
{
    vector <vector<T>> 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) {
        row = r; col = c;
        for (int i = 0; i < row; i++) {
            vector<T> t;
            p_data.push_back(t);
            for (int j = 0; j < col; j++) {
                T temp;
                p_data[i].push_back(temp);
            }
        }
    }

    ~Matrix() {
        p_data.clear();
    }

    vector<T>& operator[] (int i) {
        return p_data[i];
    }

    Matrix& operator = (const Matrix& m) {
        if (row != m.row || col != m.col) {
            p_data.clear();
            row = m.row; col = m.col;
            for (int i = 0; i < row; i++) {
                vector<T> t;
                p_data.push_back(t);
                for (int j = 0; j < col; j++) {
                    T temp;
                    p_data[i].push_back(temp);
                }
            }
        }
        for (int i = 0; i < row; i++)
            for (int j = 0; j < col; j++)
                p_data[i][j] = m[i][j];
        return *this;
    }
    bool operator == (const Matrix& m) const {
        if (row != m.row || col != m.col) return false;
        for (int i = 0; i < row; i++)
            for (int j = 0; j < col; j++)
                if (p_data[i][j] != m[i][j]) return false;
        return true;
    }
    Matrix operator + (const Matrix& m) const {
        Matrix t(row, col);
        for (int i = 0; i < row; i++)
            for (int j = 0; j < col; j++)
                t[i][j] = p_data[i][j] + m.p_data[i][j];
        return t;
    }

    Matrix operator * (const Matrix& m) const {
        Matrix t(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++)
                    if (k == 0) t[i][j] = p_data[i][k] * m.p_data[k][j];
                    else
                        t[i][j] = t[i][j] + p_data[i][k] * m.p_data[k][j];
        return t;
    }
    friend std::ostream& operator<<(std::ostream& o, const Matrix<T>& c) {
        for (int i = 0; i < c.row; i++){
            for (int j = 0; j < c.col; j++){
                o << c.p_data[i][j] << " ";
            }
            o << "\n";
        }
        return o;
    }


};

template <class T>
istream& operator >> (istream& input, Matrix<T>& M) {
    int n, m;
    input >> n >> m;
    Matrix<T>t(n, m);
    M = t;
    for(int i=0;i<M.row;i++){
        for (int j = 0; j < M.col; j++) {
            input >> M[i][j];
        }
    }
    return input;
}
template <class T>
ostream& operator << (ostream& output, const Matrix<T>& M) {
    output << M.row << " " << M.col << endl;
    for (int i = 0; i < M.row; i++) {
        for (int j = 0; j < M.col; j++) {
            output << M[i][j];
            output << " ";
        }
        output << endl;
    }
    return output;
}
```

测试用例：

```
int main() {
    //int
    Matrix<int> intMatrix1(3, 2);
    Matrix<int> intMatrix2(3, 2);
    Matrix<int> intMatrix3(2, 3);
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 2; ++j) {
            intMatrix1[i][j] = i + j;
            intMatrix2[i][j] = i + j;
            intMatrix3[j][i] = i + j;
        }
    }
    cout << "M1 + M2:\n";
    cout << intMatrix1 + intMatrix2;
    cout << "M3 * M2:\n";
    cout << intMatrix3 * intMatrix2;
    // Complex
    Matrix<Complex> complexMatrix1(3, 2);
    Matrix<Complex> complexMatrix2(2,3);
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 2; ++j) {
            complexMatrix1[i][j] = Complex(i,j);
            complexMatrix2[j][i] = Complex(j,i);
        }
    }
    cout << "CM2 * CM1:\n";
    cout << complexMatrix2 * complexMatrix1;
    return 0;
}
```

测试结果：

```
M1 + M2:
0 2
2 4
4 6
M3 * M2:
5 8
8 14
CM2 * CM1:
5i -3+5i
3+5i 8i
```