概念题

1. 简述 C++中类属的概念.
   程序实体能对多种类型的数据进行操作或描述的特性称为类属或泛型

2. C++提供了哪两种实现类属函数的机制? 简述它们的缺点.
   采用通用指针类型的参数: 需要大量的指针操作, 比较麻烦;
      编译程序无法进行类型检查, 容易出错;
   函数模板: 模板的复用会导致多模块的编译结果中存在相同的实例:
      相同的函数模板实例, 相同的类模板成员函数实例.
      相同代码的存在会造成目标代码庞大.

3. 简述 C++中参数化多态的概念及作用
   带有类型参数的代码, 给参数提供不同的类型得到不同的代码.
   使得一段代码可以有多种解释, 通过实例化可以对源代码进行复用.

编程题

1. 猜想以下每个调用会匹配到哪个函数模板, 以及对应的 T 是什么数据类型.
   f(a): 匹配到 void f(T), 对应的 T 是 int 型
   f(b): 匹配到 void f(T), 对应的 T 是 int*型
   f(c): 匹配到 void f(T), 对应的 T 是 const int 型
   f(d): 匹配到 void f(const T*), 对应的 T 是 int 型
   g(a): 匹配到 void g(T), 对应的 T 是 int 型
   g(b): 匹配到 void g(T*), 对应的 T 是 int 型
   g(c): 匹配到 void g(T), 对应的 T 是 const int 型
   g(d): 匹配到 void g(T*), 对应的 T 是 const int 型

2. 实现任意类型数据的单向链表, 创建节点类模板 Node 和单向链表类模板 List
   通过 List 可创建一个空链表, void add(T dat)在链表末尾添加一个新节点
   void display()顺序输出所有节点的 T 类型数据, 生命周期结束自动释放节点所占空间
   在 main 函数中使用链表类模板 List 建立三条链表, 分别添加若干节点并显示数据

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

template <class T>
struct Node {
    T value;
    Node* next;
};

template <class T>
class List {
```

```
    Node <T>* head;
    Node <T>* tail;
public:
    List() { head = NULL; tail = NULL; }
    void add(T dat);
    void display();
    ~List();
};

template <class T>
void List <T>::add (T dat){
    Node <T>* temp = new Node <T>;
    temp->value = dat;
    temp->next = NULL;
    if (head == NULL) {
        head = temp;
        tail = temp;
    }
    else {
        tail->next = temp;
        tail = temp;
    }
}

template <class T>
void List <T>::display() {
    Node <T>* p;
    if (head != NULL) {
        for (p = head; p->next != NULL; p = p->next) {
            cout << p->value << ' ';
        }
        cout << p->value << endl;
    }
}

template <class T>
List <T>::~List() {
    Node <T>* p, * q;
    for (p = head; p != NULL; ) {
        q = p;
        p = p->next;
        q->next = NULL;
        delete q;
    }
```

```
}

int main()
{
    List<int> lst1;
    lst1.add(1);
    lst1.add(2);
    lst1.add(3);
    lst1.display();
    List<double> lst2;
    lst2.add(0.1);
    lst2.add(0.2);
    lst2.add(0.3);
    lst2.display();
    List<string> lst3;
    lst3.add("Hello");
    lst3.add("Hi");
    lst3.add("Goodbye");
    lst3.display();
    return 0;
}
```

3. 设矩阵类模板 Matrix 可对元素值为任意类型数据的矩阵进行如下操作.

1) Matrix(int m,int n)创建 m 行 n 列矩阵, Matrix(const Matrix &mat)用 mat 拷贝创建
   void setMatrix()给所有矩阵元素赋值, void display()打印矩阵
   void transport()用矩阵的转置替换原矩阵
   完成矩阵类模板, 创建两个 T 为 int 型的矩阵, 设计程序使用以上操作

2) 类模板 Matrix 中新增两种矩阵操作:
   void square()将矩阵与自身的转置相乘的结果打印出来
   Matrix operator+(const Matrix &mat)重载运算符"+"
   创建三个 T 为 double 型的同大小矩阵, 设计程序使用这两种操作

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

template <class T>
class Matrix {
    int row;
    int col;
    T** matrix;
public:
    Matrix(int r, int c) {
        row = r;
        col = c;
```

```
        matrix = new T * [row];
        for (int i = 0; i < row; i++) {
            matrix[i] = new T[col];
            memset(matrix[i], 0, col);
        }
    }
    Matrix(const Matrix& mat) {
        row = mat.row;
        col = mat.col;
        matrix = new T * [row];
        int i, j;
        for (i = 0; i < row; i++) {
            matrix[i] = new T[col];
            for (j = 0; j < col; j++)
                matrix[i][j] = mat.matrix[i][j];
        }
    }
    void setMatrix() {
        int i, j;
        T temp;
        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                cin >> temp;
                matrix[i][j] = temp;
            }
        }
    }
    void display() {
        int i, j;
        for (i = 0; i < row; i++) {
            for (j = 0; j < col - 1; j++) {
                cout << matrix[i][j] << ' ';
            }
            cout << matrix[i][col - 1] << endl;
        }
    }
    void transport() {
        Matrix<T> temp(*this);
        for (int i = 0; i < row; i++)
            delete[] matrix[i];
        delete[] matrix;
        matrix = NULL;
        int t = row;
        row = col;
```

```
        col = t;
        matrix = new T * [row];
        for (int i = 0; i < row; i++) {
            matrix[i] = new T[col];
            memset(matrix[i], 0, col);
        }
        int i, j;
        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                matrix[i][j] = temp.matrix[j][i];
            }
        }
    }
    void square() {
        T temp = matrix[0][0];
        int i, j;
        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                temp += matrix[i][j] * matrix[i][j];
            }
        }
        temp -= matrix[0][0];
        cout << temp << endl;
    }
    Matrix operator+(const Matrix& mat) {
        Matrix<T> temp(*this);
        int i, j;
        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                temp.matrix[i][j] += mat.matrix[i][j];
            }
        }
        return temp;
    }
    ~Matrix() {
        for (int i = 0; i < row; i++)
            delete[] matrix[i];
        delete[] matrix;
    }
};

int main()
{
    Matrix<int> m1(3, 4);
```

```
    m1.setMatrix();
    m1.display();
    Matrix<int> m2(m1);
    m2.display();
    m2.transport();
    m2.display();

    Matrix<double> m3(2, 3);
    m3.setMatrix();
    m3.display();
    m3.square();
    Matrix<double> m4(2, 3);
    m4.setMatrix();
    m4.display();
    Matrix<double> m5 = m3 + m4;
    m5.display();
    return 0;
}
```