

第三次作业

概念题

一. C++ 中的成员对象是指什么？创建包含成员对象的类的对象时，构造函数和析构函数的调用次序是怎样的？

- 成员对象定义：对于类的数据成员，其类型可以是另一个类。当一个类**A**的对象**m**作为另一个类**B**的数据成员时，**m**被称为**B**的成员对象。
- 构造函数调用次序：
 - 先执行成员对象类的构造函数，再执行本身类的构造函数。
 - 若一个类包含多个成员对象，则这些成员对象按他们在类中的说明顺序进行初始化。
- 析构函数调用次序：
 - 与构造函数调用次序相反。
 - 先执行本身类的析构函数，再执行成员对象类的析构函数。
 - 若一个类包含多个成员对象，则这些成员对象按他们在类中的说明顺序的逆序执行析构函数。

二. 在哪些情况下，会调用类的拷贝构造函数？什么时候需要自定义拷贝构造函数，为什么？

- 调用情况：
 - 创建一个对象且用同类型的对象对其初始化时调用。
 - 把对象作为值参数传给函数时调用。
 - 把对象作为函数的返回值时调用。
- 定义自定义拷贝构造函数的情况：
 - 当类的对象会额外申请一些资源时，默认的拷贝构造函数不会拷贝这些资源，这会导致新对象与原对象共享资源，在资源使用和释放上会出现问题。若不是有意为之，则需要自定义拷贝构造函数。
 - 隐式拷贝构造函数逐个成员拷贝初始化，若希望有其他的操作，则需要自定义拷贝构造函数。

三. 请说明 C++ 中 **const** 和 **static** 关键字的作用。

- **const**:
 - **const**修饰一般变量，则该变量无法被改变。
 - **const**修饰引用，可以使得无法通过引用修改其引用的变量。
 - **const**修饰指针
 - 当**const**临近类型，如`const int* a`, `int const* a`，则不能通过该指针修改其执行变量的值
 - 当**const**不临近类型，如`int* const a`，则指针不能指向其他的变量。
 - **const**用于参数传递和返回值，使得传入的参数和返回值不被改变。
 - **const**修饰成员函数，则不能通过该成员函数修改对象的状态。
 - **const**修饰对象，则该对象只能调用常成员函数，不能调用非常成员函数。
- **static**
 - 扩展生存期
 - 将局部变量的自动生存期扩展为静态生存期。
 - 限制作用域
 - 将全局变量的全局作用域限制到文件作用域。

- 唯一性
 - 将类的数据成员由属于单个对象变为属于整个类。
 - `static`修饰数据成员，用于在数据抽象和数据封装原则下实现同一个类的不同对象间共享数据。
 - `static`修饰成员函数，可以限制其只能访问类的静态成员。

四. 简述C++ 中友元的概念、友元的特性以及友元的利弊。

- 概念：定义在该类外，但可以直接访问该类的`private`和`protected`成员的程序实体（某些全局函数、某些其它类或某些其它类的某些成员函数）。
- 特性：
 - 不对称性。例如：假设B是A的友元，如果没有显式指出A是B的友元，则A不是B的友元。
 - 非传递性。例如：假设B是A的友元、C是B的友元，如果没有显式指出C是A的友元，则C不是A的友元。
 - 与一个类密切相关的、又不适合作为该类成员。
- 利：避免为了获得数据成员而进行反复的函数调用，提高了访问效率。
- 弊：一定程度上违背了类的封装，数据保护的思想，降低了安全性。

编程题

一.

- 源代码的运行结果：

```
Matrix
~Matrix 0xee1830
~Matrix 0xee1830
```

- 源代码存在的问题：
- 类中`m_data`指针动态申请了额外的空间，但没有自定义拷贝构造函数，导致`m2`的`m_data`指针与`m1`的`m_data`指针指向同一块内存，在析构函数中释放堆区空间时重复释放，产生错误。
- 改进后的代码：

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

class Matrix{
private:
    int dim;
    double *m_data;
public:
    Matrix(int d);
    Matrix(const Matrix& m);    //自定义拷贝构造函数
    ~Matrix();
};

Matrix::Matrix(int d){
    dim = d;
```

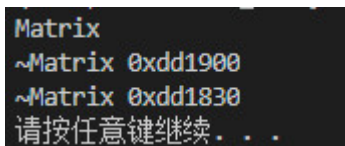
```

        m_data = new double[dim*dim];
        cout << "Matrix" << endl;
    }

    Matrix::~Matrix(){
        cout << "~Matrix " << m_data << endl;
        delete [] m_data;
        m_data = NULL;
    }
    /**
     * 改进部分，自定义拷贝构造函数
     */
    Matrix::Matrix(const Matrix& m){
        dim = m.dim;                //普通变量直接复制
        m_data = new double[dim * dim]; //重新申请内存区域
    }
    int main(){
        {
            Matrix m1(5);
            Matrix m2(m1);
        }
        system("pause");
        return 0;
    }

```

- 改进后的运行结果：



```

Matrix
~Matrix 0xdd1900
~Matrix 0xdd1830
请按任意键继续...

```

- 正确性说明：
 - 在自定义的拷贝构造函数内为m2的m_data重新申请了相同大小的堆区内存，使得m1的m_data和m2的m_data指针不指向同一块内存，避免了释放时的错误，也完成了拷贝构造的要求。

二.

```

#include <iostream>
using namespace std;

class Shooting{
    float FTPercentage; // 罚球命中率
    float FGPercentage; // 投篮命中率
    float TPPercentage; // 三分命中率
public:
    Shooting(){
        FTPercentage = 0.2;
        FGPercentage = 0.2;
    }

```

```

        TPPercentage = 0.2;
    }
    Shooting(float ftp, float fgp, float tpp){
        FTPercentage = ftp;
        FGPercentage = fgp;
        TPPercentage = tpp;
    }
};

class NBAPlayer{
    Shooting shoot; // 实例化 Shooting 对象 shoot
    string name;
public:
    //补全1. 调用 Shooting 的默认构造函数对 shoot 初始化;
    NBAPlayer(string name){
        this->name = name;
    }
    //补全2. 调用 Shooting(float ftp, float fgp, float tpp) 构造函数对 shoot 初始化;
    NBAPlayer(string name, float ftp, float fgp, float tpp):shoot(ftp, fgp, tpp){
        this->name = name;
    }
};

int main(){
    //补全3. p1.name 初始化为 Curry, p1.shoot的各项命中率采用默认初始化;
    NBAPlayer p1("Curry");
    //补全4. p2.name 初始化为 Curry, p2.shoot的 FTPercentage初始化为
    0.9,FGPercentage 初始化为 0.71, TPPercentage 初始化为0.44
    NBAPlayer p2("Curry", 0.9, 0.71, 0.44);
    return 0;
}

```

三.

```

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

class Component {
private:
    int Weight;
public:
    static int TotalWeights;
    Component(int w){
        Weight = w;
        TotalWeights += w; //增加该零件的重量
    }
    int GetWeights(){
        return this->Weight;
    }
};

int Component::TotalWeights = 0; //静态成员初始化

```

```

int main(int argc, char const *argv[]){
    int a, b;
    cin>>a>>b;
    Component A(a);
    Component B(b);
    cout<<B.TotalWeights<<endl;
    return 0;
}

```

四.

```

#include<iostream>
#include<string>
using namespace std;
class Lion;
class Tiger;
int totalWeights(const Lion& L, const Tiger& T);

class Lion {
private:
    int weight;
public:
    Lion(int w){
        weight = w;
    }
    friend int totalWeights(const Lion& L, const Tiger& T);    //友元函数声明
};

class Tiger {
private:
    int weight;
public:
    Tiger(int w){
        weight = w;
    }
    friend int totalWeights(const Lion& L, const Tiger& T);    //友元函数声明
};

int main(int argc, char const *argv[]){
    int w1, w2;
    cin>>w1>>w2;
    Lion L(w1);
    Tiger T(w2);
    cout<<totalWeights(L, T)<<endl;
    return 0;
}

int totalWeights(const Lion& L, const Tiger& T){
    return L.weight + T.weight;
}

```

