课堂内容复习演练

观察临时对象

观察一行之间发生了什么

```
#include <cinttypes>
#include <iostream>
#include <functional>
#include <variant>

using namespace std;

#define PRINTFUN_ADDR std::cout << static_cast<const char *>(__PRETTY_FUNCTION__)
<< " at " << this << std::endl

class A{
    int m_a = 1;
    int m_b = 2;
    int m_c = 3;
    uint8_t m_d = 4;
public:
    A(int a, int b, int c, uint8_t d);
    A();

    A(const A& rhs): m_a(rhs.m_a), m_b(rhs.m_b), m_c(rhs.m_c), m_d(rhs.m_d)
    {
        PRINTFUN_ADDR;
    }
    ~A() { PRINTFUN_ADDR; };
    void Dump();
    void DoSomething()
    {
        cout << "Something" << endl;
    }
};

void A::Dump()
{
     using namespace std;
     auto dumpItem = [this](auto name, auto& item){
        std::cout << name << "@" << (void*)&item << ", value is " << item <<  ",
size is " << sizeof(item) << endl;
     };
     dumpItem("m_a", m_a);
     dumpItem("m_b", m_b);
     dumpItem("m_c", m_c);
     dumpItem("m_d", m_d);
     cout << "Obj address:" << sizeof(*this) << endl;
}

A::A(int a, int b, int c, uint8_t d): m_a(a), m_b(b), m_c(c)
```

```
{
    PRINTFUN_ADDR;
}

A::A()
{
    PRINTFUN_ADDR;
}

void Func(A item)
{
    item.DoSomething();
}

int main()
{
    cout << "start show" << endl;
    A a{1,2,3,4};
cout << ">>>>" << __LINE__ << endl;
    Func(a); // A{5,6,7,8}.Dump();
cout << "<<<<<" << __LINE__ << endl;
    cout << "ending..." << endl;
}
```

godbolt

编译并观察 copy-swap 实现的行为，并和手写的方式进行比较

Day 0

小型数据输出系统

某系统原来使用一个封装的 CFileWriter 类用于数据输出到文件中，现在需要将其移植到一个不支持文件的系统上，用户希望这个系统还支持将数据写入指定的内存块中，同时，主管又提出了一个可测试性的需求，希望能够通过自动化测试手段检测系统输出的顺序和数据位置是否正确。需要设计一个小型数据输出系统，能够对原来的系统不修改或尽量少的修改就可以满足新需求。

当前已有的系统：

```
class CFileWriter
{
public:
    CFileWriter(const std::string& filename);
    CFileWriter() = delete;
    ~CFileWriter();

    int WriteAtBegin(void* data, int length) {
        std::cout << "Write " << length << " bytes at file begin" << std::endl;
        return length;
    }
    int WriteAt(int pos, void* data, int length) {
        std::cout << "Write " << length << " bytes at " << pos << std::endl;
        return length;
    }
    int WriteAtEnd(void* data, int length) {
```

```
        std::cout << "Write " << length << " bytes at file end" << std::endl;
        return length;
    }

private:

};

class CClient final
{
public:
    CClient(CFileWriter* writer) : m_writer(writer) {}
    CClient() = delete;
    ~CClient() {}

    bool DoProcess() {
        char buf[64] = { 0 };

        memset(buf, 'A', 64);
        m_writer->WriteAtBegin(buf, 64);
        memset(buf, 'B', 32);
        m_writer->WriteAt(16, buf, 32);
        memset(buf, 'C', 64);
        m_writer->WriteAtEnd(buf, 64);
    }
private:
    CFileWriter *m_writer;
};

int main()
{
    CFileWriter writer("filename");
    CClient c(&writer);

    c.DoProcess();
}
```

请根据需求思考如何设计，给出设计图，注意抽象类，子类的接口设计。

设计模式 Strategy

请按照 Strategy 模式改造 CWriter 类来扩展原有类的功能，支持多种写策略。

可能的思路:

在 CWrite 类中增加 IWriteStrategy 指针成员。

```
enum class StrategyTag
{
   NORMAL_WRITE,
   LAZY_WRITE,
   RELIABLE_WRITE,
};

class CFileWriter
{
```

```
public:
    CFileWriter(const std::string& filename, StrategyTag strategy = NORMAL_WRITE
);
    // ...;

private:
    IWriteStrategy* m_pStrategy;
};
```

在派生类中实现 2 到 3 种策略，如：

- NORMAL_WRITE，行为跟已有设计完全一样
- LAZY_WRITE，写入的内容会暂存到内部的一个数组或容器中，只有当数组或容器将满时才真正写入
- RELIABLE_WRITE，写入的内容会回读校验，如有错，会报错或抛出异常

```
class IWriteStrategy
{
    virtual int OnWriteAt(int pos, void* data, int length);

};
```

进阶要求，尝试采用模板 policy 的方式在编译期实现。

工业级实现：

Shenandoah GC 是由 Red Hat 贡献给 OpenJDK HotSpot JVM 的一个并发的垃圾回收器。它的启发策略使用了 Strategy 模式来抽象。ShenandoahHeap 初始化时会根据命令行参数选择不同的策略，而调用策略的 ShenandoahControlThread 线程则只“知道” ShenandoahHeuristics 及其提供的接口，无需“关心”具体选择的是什么启发策略。

Shenandoah GC
shenandoahHeuristics.hpp
shenandoahHeuristics.cpp
shenandoahAdaptiveHeuristics.hpp
shenandoahAggressiveHeuristics.hpp