0. 课堂内容复习演练

在 godbolt.org 上观察实现继承、接口继承，并使用 sizeof() 观察空基类优化的规律。

```
#include <iostream>

class Empty {
   using Int = int;
public:
   //virtual ~Empty() = default;
    void Fun0() { std::cout << "func0" << std::endl; }
};

class EmptyToo : public Empty {
   void Fun1() { std::cout << "func1" << std::endl; }
};

class EmptyThree : public EmptyToo {};
class NonEmpty : public Empty, public EmptyToo {};

class EmptyTooV : virtual public Empty { void Fun0() { std::cout << "my func0" <<
std::endl; } };
class EmptyTwoV : virtual public Empty { void Fun1() { std::cout << "func1" <<
std::endl; } };
class EmptyThreeV : public EmptyTooV {};
class NonEmptyV : virtual public Empty, public EmptyTooV { };
class NonEmptyVV : public EmptyTooV, public EmptyTwoV { };

int main()
{
   std::cout << "sizeof(Empty): " << sizeof(Empty) << '\n';
   std::cout << "sizeof(EmptyToo): " << sizeof(EmptyToo) << '\n';
   std::cout << "sizeof(EmptyThree): " << sizeof(EmptyThree) << '\n';
   std::cout << "sizeof(NonEmpty): " << sizeof(NonEmpty) << '\n';
   std::cout << "sizeof(EmptyTooV): " << sizeof(EmptyTooV) << '\n';
   std::cout << "sizeof(EmptyThreeV): " << sizeof(EmptyThreeV) << '\n';
   std::cout << "sizeof(NonEmptyV): " << sizeof(NonEmptyV) << '\n';
   std::cout << "sizeof(NonEmptyVV: " << sizeof(NonEmptyVV) << '\n';

    EmptyToo b;
    const Empty* pb = &b;
    std::cout << typeid(*pb).name() << '\n';
    EmptyTooV b1;
    Empty* pb1 = &b1;
    std::cout << typeid(*pb1).name() << '\n';
    pb1->Fun0();
    std::cout << (typeid(*pb) == typeid(*pb1)) << std::endl;
}
```

gobolt

1. pImpl

请扩展 FileWriter 类的功能，为其实现真正的文件写入功能。要求使用 pImpl 习语将实现细节放在 FileWriter.cpp 内部。

```
class CFileWriter : public IWriter
{
public:
    CFileWriter(const std::string& filename, StrategyTag tag =
StrategyTag::NORMAL);
    ~CFileWriter();

    CFileWriter() = delete;
    CFileWriter(const CFileWriter&) = delete;
    CFileWriter& operator=(const CFileWriter&) = delete;
    CFileWriter(CFileWriter&&) = delete;
    CFileWriter& operator=(CFileWriter&&) = delete;

// IWriter interfaces
    int WriteAtBegin(void* data, int length) override;
    int WriteAt(int pos, void* data, int length) override ;
    int WriteEnd(void* data, int length) override ;
private:
    struct FileWriterImpl;
    const FileWriterImpl* m_pImpl;
    IWriteStrategy* m_pStrategy = nullptr;
};
```

工业级实现

请尝试将 MLIRContext 以及相关的类 Type.h 的早期实现临摹到练习项目中，让其可以单独编译，并通过一些基本测试。思考 MLIRContext 和 Type 之间的关系，pImpl 封装固化的是什么，请注意参考实现中现代 C++ 特性和库的使用。

MLIR
Init version
D-Pointer in QT

2. 观察者

- 写出 Event, Observable 和 Observer 三个类
- Observable `has-a` 一些 Event 成员, 设计接口让（任意数量的）外部 Observer 对象能够在运行期订阅和取消订阅这些 Event 并触发相应回调
- Observer 生命周期结束时要取消所订阅的 Event
- 实现成功后将上节练习中的 IWriter 的一个派生类改造成一个 Observable, 支持在写开始/完成的时候触发回调打印信息
- 将参考实现中 Event 管理订阅者容器改造为 pImpl 模式，支持以不同的容器实现管理

相关语言知识

- `has-a` vs `is-a`
- 运算符重载
- 容器的使用
- 变参模板（不要求）

参考实现

godbolt

```
#include <cstddef>
#include <functional>
#include <map>
#include <iostream>

using EventToken = size_t;
using namespace std;

template <typename... Args>
class Event
{
    public:
        virtual ~Event() = default;
         Event() noexcept = default;
         Event(const Event&) noexcept = delete;
         Event& operator=(const Event&) noexcept = delete;
         Event (Event&&) = delete;
         Event & operator=(Event&&) = delete;

        [[nodiscard]] EventToken operator+=(function<void(Args...)> observer)
        {
            auto n{++m_counter}; //why?
            m_observers[n] = observer;
            return n;
        }

        Event& operator-=(EventToken handle)
        {
            m_observers.erase(handle);
            return *this;
        }
        void raise(Args... args)
        {
            for (auto& observer : m_observers) { (observer.second)(args...);}
        }
    private:
        size_t m_counter {};
        map<EventToken, function<void(Args...)>> m_observers;

};

class Observable
{
    public:
        auto& getEventWriteStarted() { return m_eventWriteStart;}
        auto& getEventWriteEnded() { return m_eventWriteEnd;}
        void OnWriteStart(int startOffset, int size)
        {
            getEventWriteStarted().raise(startOffset, size);
        }
        void OnWriteEnd(void)
        {
            getEventWriteEnded().raise();
        }
    private:
        Event<int, int> m_eventWriteStart;
        Event<> m_eventWriteEnd;
};

void GlobalLogWriteStart(int offset, int size)
{
    cout << "Global: Write start at " << offset <<
    ", will write " << size << "byets" << endl;
 };

 class Observer
 {
     public:
        Observer(Observable& target) : m_target { target }
        {
            m_targetWriteStartHandle = m_target.getEventWriteStarted() +=
            [this](int startOffset, int size) {
                onTargetWriteStarted(startOffset, size);
            };
        }

        virtual ~Observer()
        {
            m_target.getEventWriteStarted() -= m_targetWriteStartHandle;
        }

         Observer(const Observer&) noexcept = delete;
         Observer& operator=(const Observer&) noexcept = delete;
         Observer (Observer&&) = delete;
         Observer & operator=(Observer&&) = delete;
    private:
        void onTargetWriteStarted(int startOffset, int)
        {
            cout << "Observer: I saw you write from " << startOffset << endl;
        }
        Observable& m_target;
        EventToken m_targetWriteStartHandle;
 };

int main()
{
    Observable writer;
    EventToken handleWriteStart { writer.getEventWriteStarted() +=
GlobalLogWriteStart };
    EventToken handleWriteEnd {
        writer.getEventWriteEnded() += [] {
            cout << "Lambdas: Write Ended" << endl;
        }
    };

    Observer observer{ writer};
    writer.OnWriteStart(42,1000);
    writer.OnWriteEnd();
    cout << endl;
    writer.getEventWriteStarted() -= handleWriteStart;
    writer.OnWriteStart(1024, 2);
    writer.OnWriteEnd();
}
```

工业级实现

请注意意对多线程的支持和 RAII 的使用

clangd/support/Funciton.h

clangd/unittests/support/FuntionTests.cpp

3. Template Method(课后）

基本要求

请将 FileWriter 中策略的生成改造为 Factory Method。从而将策略的生成与使用进一步隔离。

工业级实现

通过 DeviceFactory 及其纯虚函数成员 CreateDevices，将 Device 对象的构造和它的表示解耦，也解耦了如何构造和在什么时机
Tensorflow Device Factory