GameEngine1

2024年5月25日 15:04

----------项目设置----------

》》》》项目设置：

1.建立一个github库

2.设置一个VS解决方案和项目

3.设置相应的配置

4.链接

1.配置：

设置引擎为一个库文件（dll），在外部将库文件链接到外部的应用项目（exe）

（静态库的形式类似于将一大堆库链接到游戏中）

（动态库的形式类似于将一大堆外部库先链接到dll文件中，再将这个dll文件链接到游戏中，这样我们的游戏只会依赖于这一个dll文件）

1.删除了适应平台（x86）

2.改变配置类型（exe -> dll）

3.更改输出目录和中间目录

2.新建一个项目并且配置其支持平台，输出和中间目录（和引擎相同）

3.设置启动项目：

右键Sandbox并选择该选项(.vs文件会保存我们在vs中做出的配置调整，但我们要为其他平台启动的人做一些调整0

4.调整sln文件中的启动项

在文本编辑器（可以是vscode）中打开解决方案文件.sln，调整前几句为

```
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "Sandbox", "Sandbox\Sandbox.vcxproj", "{28573136-9FAB-4D60-8F24-3DF8BCC0422B}"
EndProject
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "Nut", "Nut\Nut.vcxproj", "{7FB1D529-C182-497A-A2B0-633BC7A48C81}"
EndProject
```

5.链接

将启动项目和引擎库链接，右键sandbox -> add（添加） -> reference（引用） -> 勾选你的引擎文件

----------引擎入口点----------

》》》》什么是入口点

引擎入口点（Engine Entry Point）通常指的是一个程序的起始执行位置，也可以被称为主函数（Main Function）。程序从这里开始执行，并按照预定的流程继续执行。

eg:: 例如，在C语言中，引擎入口点通常被命名为main函数，它是程序的起始位置。在C++中，引擎入口点可以是全局的main函数，也可以是类的静态成员函数。

》》》》什么是沙盒(sandbox)

sandbox（沙盒）是一种安全机制，用于限制程序的访问权限和行为范围。它创建了一个受限的执行环境，将程序隔离在其中，以防止恶意代码或不安全的操作对系统造成损害。

》》》》虚析构函数?

当一个类的析构函数被声明为虚析构函数时，这意味着该类将成为多态类型，并且可以安全地通过基类指针删除派生类对象。

1.安全地销毁派生类对象：当使用基类指针指向派生类对象时，如果基类的析构函数不是虚函数，在使用delete删除指针时只会调用基类的析构函数。这可能导致派生类中的资源泄漏，因为派生类的析构函数未被调用。
通过将析构函数声明为虚函数，可以确保在删除指向派生类对象的基类指针时，会先调用派生类的析构函数，然后再调用基类的析构函数，从而正确释放派生类所占用的资源。

2.支持多态行为：在使用基类指针指向派生类对象并调用虚函数时，会根据对象的实际类型来调用相应的函数。

》》》》实现思路

通过应用程序是否执行任务（比如是否在windows平台，是否加载了一个dll文件...）来进行条件判断，也就是捕获了这些事件，然后利用这个条件运行某段代码。

》》》》类的继承

在 C++ 中，有三种继承方式：公有继承（public inheritance）、私有继承（private inheritance）和受保护继承（protected inheritance）。
它们的区别在于派生类对基类成员的访问权限。

公有继承（public inheritance）：

语法：使用 public 关键字进行声明，例如 class 派生类 : public 基类 {}。

基类的公有成员在派生类中仍然是公有的。

基类的保护成员在派生类中仍然是保护的。

基类的私有成员在派生类中不可访问。

私有继承（private inheritance）：

语法：使用 private 关键字进行声明，例如 class 派生类 : private 基类 {}。

基类的公有成员在派生类中变为私有的。

基类的保护成员在派生类中变为私有的。

基类的私有成员在派生类中不可访问。

受保护继承（protected inheritance）：

语法：使用 protected 关键字进行声明，例如 class 派生类 : protected 基类 {}。

基类的公有成员在派生类中变为受保护的。

基类的保护成员在派生类中仍然是受保护的。

基类的私有成员在派生类中不可访问。

选择继承方式应根据具体的设计需求和情况来决定。
通常情况下，公有继承是最常用的继承方式，因为它能够使派生类获得基类的接口和功能，并且符合面向对象编程的封装性和多态性原则。
私有继承和受保护继承在特定场景下有其用途，比如实现继承实现细节封装或限制派生类对基类接口的访问。

----------引擎日志----------

》》》》

思路：使用C#风格的库spdlog，将其创建为子模版。将spdlog接口打包，方便使用。
打包后将其设计为宏函数，方便使用。也方便在发行时候通过 #ifdef 来控制一系列宏函数打印的日志不用在发行版本使用。

》》》》子模版？

git submodule add 命令会在主仓库中创建一个指向子模块仓库的链接，并将子模块仓库克隆到指定的目录下。
这个链接存储在主仓库的 .gitmodules 文件中，以便记录和管理子模块的相关信息。

通过将外部依赖库作为子模块添加到主仓库中，你可以保持主仓库和子模块仓库的独立性。
这意味着主仓库和子模块仓库可以分别进行版本控制和更改，而不会相互干扰。
当你在不同的项目中使用相同的外部依赖库时，你只需要在这些项目中添加子模块的链接，而不必重复复制和维护这些外部依赖库的副本。

----------引擎脚本(use Premake API)----------

》》》》思路：

使用premake内置的API接口编写premake.lua脚本文件，使其自动构建特定于平台的项目文件，并自动化的完成Dll文件的复制--替换操作。
同时通过bat文件自动化输入命令启动premake5.exe文件的这个操作。

----------事件系统设计----------

》》》》设计：

四个事件文件
　　-->AppEvent
Event.h -->KeyEvent
　　-->MouseEvent

》》》》代码解析：

1.EventType::##type 和 #type 是什么？

EventType::##type 中的 ## 是预处理操作符，用于将宏参数 type 与 EventType:: 连接起来。例如，如果 type 是 Mouse，那么 EventType::##type 就会被展开为 EventType::Mous e。
return #type; 中的 # 是字符串化操作符，将宏参数转换为字符串。如果 type 是 Mouse，那么 #type 就会被展开为 "Mouse"。

2.宏定义 EVENT_CLASS_TYPE(type) 的逻辑是什么？

EVENT_CLASS_TYPE 宏定义了三个函数：GetStaticType、GetEventType 和 GetName。其中 GetStaticType 返回事件的静态类型，即将 type 参数与 EventType:: 连接。GetEvent Type 实际上调用了 GetStaticType 函数。GetName 返回事件对象的名称，即将 type 参数转换为字符串。

3.枚举类型 EventCategory 和位运算的使用是怎样的？

在代码中，使用 #define BIT(x) (1 << x) 定义掩码常量，表示对应位置为 1。而 EventCategory 枚举类型定义了五种事件类别。通过位运算 & 将 category 和该事件对象所属的类别进行比较，判断该事件对象是否包含在指定事件类别中。

```
enum EventCategory
{
    None = 0,
    Mouse = 0b00000001,    // 表示鼠标事件的掩码常量
    Keyboard = 0b00000010, // 表示键盘事件的掩码常量
    Window = 0b00000100     // 表示窗口事件的掩码常量
};
```

假设 GetCategoryFlags() 返回的是 Mouse 类别的掩码常量 0b00000001，
而 category 是另一个掩码常量，例如 Keyboard 类别的掩码常量 0b00000010。
那么当二者进行按位与运算时，结果如下所示：

```
0b00000001  (GetCategoryFlags() 的值，Mouse 类别的掩码常量)
&
0b00000010  (category 的值，Keyboard 类别的掩码常量)
----------
00000000   (结果为 0，表示不属于 Keyboard 类别)
```

因此，结果是一个新的值，其比特位是根据两个操作数的相应比特位进行按位与操作得到的。
在这个例子中，结果为 0，表示不属于键盘事件类别。

4.m_Event.GetEventType() == T::GetStaticType() 的作用是什么？

这段代码是在 EventDispatcher 类中定义的模板函数 Dispatch 中，用于根据事件类型分发事件处理函数。该模板函数可以接受一个函数对象 func，该对象的参数类型为 T&。在函数体内，判断传入的事件处理函数类型是否与当前事件对象的类型匹配。

5.m_Event.m_Handled = func(*(T*)&m_Event) 的作用是什么？

这段代码在 Dispatch 模板函数中，将事件对象转换为指定类型 T 后，调用传入的处理函数 func 来处理事件，并将处理结果存储在 m_Event.m_Handled 中，标记事件是否被处理。*(T*)&m_Event 表示强制将 m_Event 转换为 T 类型的引用，并将其作为参数调用函数对象 func。

6.template<typename T> using EventFn = std::function<bool(T&)>; 和 std::function 是什么？

EventFn 是一个别名模板，定义了一个函数对象类型 std::function<bool(T&)>，表示接受一个参数类型为 T&，返回类型为 bool 的函数对象。std::function 是一个通用的函数封装类，用于封装可调用对象，如函数指针、成员函数指针、Lambda 表达式等。

》》》》enum 和 enum class 的区别
enum class 中的成员在使用时候有类名这个作用域的限制，enum 则没有

》》》》什么是事件分发器
概念：事件分发器（Event Dispatcher）是一种设计模式，用于处理和分发事件（Event）的机制。
包括以下几个要点：
接收事件：事件分发器需要能够接收系统中产生的各种事件，如按键输入、鼠标点击、网络消息等。
分发事件：根据事件的类型和属性，事件分发器将事件分发给注册的事件处理函数或对象。
事件处理：事件处理函数负责对接收到的事件做出相应的处理，可能包括更新系统状态、触发其他操作等。

eg.一个简单的事件分发器的例子是一个图形界面应用程序，当用户点击按钮时，按钮控件会生成一个点击事件，事件分发器接收到该事件后，会将事件分发给注册的按钮点击事件处理函数，从而执行按钮点击后的相应操作，比如显示弹窗、切换界面等。

》》》》std::to_string() 和 std::stringstream ss 的 ss.str()
std::to_string 函数只接受基本数据类型（例如 int、float 等）作为参数，并将其转换为 std::string 类型的字符串。
因此，直接将 std::stringstream 对象作为参数传递给 std::to_string 函数是不可行的，编译器会报错。

》》》》WindowResizeEvent WRE(1280, 720);
NUT_TRACE(WRE);
为什么能将WRE作为字符型的类型的参数传入NUT_TRACE这个宏中，并让其中的 trace( ) 函数接受WindowResizeEvent类中ToString函数的结果并输出日志?

回答：
在很多日志库中，当你将一个自定义类型的对象传递给日志输出函数时，
它们会通过调用该类型的特定方法（通常称为 ToString() 或类似的方法）来获取对象的字符串表示形式，然后将其输出到日志中。

在你的代码中，NUT_TRACE 宏展开后会调用 logger 对象的 trace 函数，并将传入的参数作为日志消息。
而在 trace 函数内部，由于传入的参数是一个 WindowResizeEvent 对象，
因此会调用 WindowResizeEvent 类中的 ToString() 方法来获取该对象的字符串表示形式。

所以，当你传递 WRE 对象给 NUT_TRACE 宏时，实际上是调用了 WRE.ToString() 方法，该方法返回一个描述 WindowResizeEvent 对象内容的字符串。
然后，这个字符串将被传递给 logger 对象的 trace 函数，并最终输出到日志中。

这种做法的好处是，可以灵活地将自定义类型的对象转换为字符串，并将其记录在日志中。

》》》》子类的构造函数中是否应该调用父类的构造函数?

在 C++ 中，如果子类构造函数没有显式调用父类构造函数，则会自动调用父类的默认构造函数（如果存在）。
以确保从父类继承而来的部分能够正确初始化，保证整个对象的完整性和正确性。

如果父类没有无参的默认构造函数，而只有带参数的构造函数，则子类必须通过初始化列表显式调用父类的构造函数来初始化从父类继承而来的部分。

eg.
```
class Base {
public:
  Base(int value) {
    std::cout << "Base constructor with value: " << value << std::endl;
  }
};

class Derived : public Base {
public:
  // 派生类构造函数没有显式调用基类构造函数
  Derived(int value) {
    // 派生类构造函数体
  }
};
```
在这个例子中，基类 Base 定义了带参数的构造函数 Base(int value)，而派生类 Derived 的构造函数没有显式调用基类构造函数。

接下来，如果我们尝试使用派生类 Derived 来创建对象：
Derived d(5);
派生类构造函数没有显式调用基类构造函数，编译器会自动尝试调用基类的默认构造函数。但是这个基类 Base 并没有默认构造函数，
因此编译器会报错指出找不到默认构造函数来初始化基类的部分。

为解决问题，可以通过初始化列表显式调用基类构造函数来初始化从基类继承来的部分：
```
class Derived : public Base {
public:
  Derived(int value) : Base(value) {
    // 派生类构造函数体
  }
};
```
修正后，我们在派生类的构造函数初始化列表中显式调用了基类的构造函数，并传递了合适的参数来初始化基类的部分。
这样就能够正确地初始化从基类继承而来的部分，避免了编译错误。

----------预编译头文件----------

》》》》理解

在premake中做出的项目设置实际上等同于在VS可视化界面上的设置

pch.h：（Use/Yc）

pch.cpp：（Create/Yc）

----------窗口和GLFW----------

》》》》fork（分支/派生）和submodule（子模块）

Fork= 就像是你复制了一个完整的项目到你自己的账号下，你可以在这个复制的项目上做任何修改而不影响原始项目。

你可以把这个复制的项目当作你自己的项目来管理。

Submodule= 就像是在一个项目中引入了另一个项目，但它们是独立的。主项目知道子项目的存在并能够与之交互，但它们是分开管理的。

子模块通常用于将一个项目作为另一个项目的一部分来使用。

简而言之，Fork 是复制整个项目到你自己的账号下，而 Submodule 是在一个项目中引入另一个项目作为子项目。

cherno的做法是：

1. 在 GitHub 上 Fork 了 glfw 库，获得自己的独立副本。
2. 向库中上传自己的premake文件。
2. 将这个 Fork 的 glfw 库作为子模块引入到自己的项目中，以便在项目中依赖和使用 glfw 库。

》》》》在查证过程中，发现cherno在当时使用的是3.3发布版本的一个开发分支。（https://github.com/TheCherno/glfw/tree/53c8c72c676ca97c10aedfe3d0eb4271c5b23dba）

位于（https://github.com/glfw/glfw/tree/53c8c72c676ca97c10aedfe3d0eb4271c5b23dba）

我选择先Fork最新的glfw，如果有其他情况以后再修正。

》》编译问题参考：

（ http://t.csdnimg.cn/SWd5W ）

It will help you a lot, believe me.

》》》》lua中的语法

1.IncludeDir = {} 是创建了一个空的 Lua 表（table），用来存储不同模块或库的包含目录。

2.而 IncludeDir["GLFW"] 则是使用了 Lua 中的表索引操作，将键为 "GLFW" 的元素设置为 "Hazel/vendor/GLFW/include"。

3.%{IncludeDir.GLFW}表示要获取表 IncludeDir 中键为 "GLFW" 的元素值

》》》》glfwInit（）

通常情况下，glfwInit() 函数会返回一个整数值来指示初始化是否成功。

》》》》glfwSetWindowUserPointer( )

作用：将一个指向自定义数据的指针与 GLFW 窗口相关联

解释：

通过调用 glfwSetWindowUserPointer(m_Window, &m_Data); 函数，你将自定义数据 m_Data 与 GLFW 窗口 m_Window 相关联。这样做的目的通常是为了在程序中可以方便地访问和操作与该窗口相关的自定义数据。例如，当你需要在 GLFW 窗口回调函数中访问特定窗口的自定义数据时，可以使用 glfwGetWindowUserPointer(m_Window) 来获取该数据指针。

》》》》glfwSetWindowUserPointer 和 glfwGetWindowUserPointer的关系和用法

void glfwSetWindowUserPointer(GLFWwindow* window, void* pointer)

参数：

window：用于设置用户指针数据的窗口对象。

pointer： 想要关联的自定义指针数据（通常是一个结构体指针或其他数据类型的指针。）

功能：将用户自定义的指针数据与特定窗口对象关联起来。方便后续取出使用。

void* glfwGetWindowUserPointer(GLFWwindow* window)

参数：

window：想要获取用户指针数据的窗口对象。

返回值：

与窗口对象关联的，用户指明的 自定义指针数据（即上面关联进来的那个数据或结构体）。

注意：

返回值是一个void *，可以指向任何数据。所以在使用时也许需要你将返回值强制类型转换并赋值给其他变量。

功能：从特定窗口对象中获取之前通过 glfwSetWindowUserPointer 设置的用户自定义指针数据。

eg.

```
// 在初始化窗口时将自定义数据与窗口对象关联
MyData data;
glfwSetWindowUserPointer(window, &data);

// 在需要时从窗口对象中获取自定义数据
MyData* userData = static_cast<MyData*>(glfwGetWindowUserPointer(window));
if (userData) {   // 使用 userData 中的数据   }
```

----------窗口事件----------

》》》》一些涉及到的知识点:

lambda:

(https://www.bilibili.com/video/BV1mw41187Ac/?p=12&vd_source=64ca0934a8f5ef66a21e8d0bddd35f63)

std::placeholders:: 1:

是 C++ 标准库中定义的占位符，用于表示函数对象中的第一个参数。用于等待下次使用时在此占位符位置上填入的值。

（这里的placeholders::1的1好像只是标明占位符的，无其他意义，比如同时使用了两个占位符那第二个占位符就是placeholders::2,

其中数字与其使用时的位置和方法没有关系，仅仅代表占位符的标号）

std::bind:

std::bind 在实际使用中有多种用途。

1.延迟调用和参数绑定

2.改变函数签名

3.成员函数绑定

4.函数适配器

```
eg.
----std::bind 延迟调用一个函数，

#include <functional>
#include <iostream>

void delayedFunction(int a, int b) {
    std::cout << "Delayed function called with arguments: " << a <<
}

int main() {
   auto delayedFunc = std::bind(delayedFunction, 10, 20);
   // 延迟执行 delayedFunction，参数被预先绑定为 10 和 20
   // ...
   // 在需要的时候调用 delayedFunc
   delayedFunc();
   return 0;
}

----改变函数的签名，包括修改函数的参数类型或个数。
    #include <functional>
    #include <iostream>
    void originalFunction(int a, int b) {
        std::cout << "Original function called with arguments: " <<
    }

void modifiedFunction(double x, double y) {
    std::cout << "Modified function called with arguments: " << x <
}
int main() {
  // 使用 std::bind 将 modifiedFunction 的签名修改为接受两个 double 类
 auto modifiedFunc = std::bind(modifiedFunction, std::placeholders:
  // 在需要的时候调用 modifiedFunc，并传入合适类型的参数
  modifiedFunc(3.14, 2.71);      // 输出：Modified function called
  return 0;
}

----绑定类的成员函数，并指定对象实例作为第一个参数。
#include <functional>
#include <iostream>

class MyClass {
public:
    void memberFunction(int value) {
        std::cout << "Member function called with value: " << value
    }
};

int main() {
    MyClass obj;
    auto memberFunc = std::bind(&MyClass::memberFunction, &obj, std
    // 绑定 MyClass 的成员函数 memberFunction，并将 obj 作为对象实例
    // ...
    // 在需要的时候调用 memberFunc
    memberFunc(42);

   return 0;
}


----使用函数适配器进行参数绑定：
    #include <functional>
    #include <iostream>

void printSum(int a, int b) {
  std::cout << "Sum: " << a + b << std::endl;
}
int main() {
   auto sumFunc = std::bind(printSum, std::placeholders::_1, 5);
   // 将第二个参数绑定为 5，等待传入第一个参数
   // ...

  // 在需要的时候调用 sumFunc
   sumFunc(10); // 输出 Sum: 15

   return 0;
}
```

----------接下来我以发问的方式来查证疑惑（这都是我在学习时产生的疑惑）----------

》》》问题：Application.cpp中的语句m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent));在干什么？

在WindowsWindow.h中，有 inline void SetEventCallback(const EventCallbackFn& callback) override { m_Data.EventCallback = callback; } 这样的定义。
所以 SetEventCallback 这个函数需要接受一个 EventCallbackFn 类型的函数，也就是 void XXX(Event& e) 这样的函数。
而std::bind恰好能返回一组函数指针或者说一个函数对象，通过这个函数对象，我们可以用传入的 OnEvent 这个函数初始化 m_Data.EventCallback
（注意：在将成员函数作为函数对象传递时，需要绑定其对象，确保能通过对象正确的访问到这个成员函数）

而 BIND_EVENT_FN(OnEvent) 就像是对 OnEvent 做了一些暂缓的设置，以便之后处理
（我们后面会谈到）

》》关于函数指针：
https://www.bilibili.com/video/BV1254y1h7Ha/?vd_source=64ca0934a8f5ef66a21e8d0bddd35f63

》》》问题：占位符呢？
虽然有占位符的设计，但是m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent));这个OnEvent却没有填入参数
即使在绑定时没有显式地填入参数，但通过占位符的机制，函数对象仍然能够正确地接收事件参数并传递给 OnEvent 函数。
（注意是 std::placeholders::_1 而不是 std::placeholders::1， 有下划线）

理解：
通过使用占位符，函数对象会暂时（注意：暂时）保留一个位置用于接收后续传入的参数，并在调用时将这些参数正确地传递给被绑定的成员函数。

》》》m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent)); 是什么意思？
BIND_EVENT_FN(OnEvent)，代表了什么意思？
随后的data.EventCallback(event); 和以上有什么联系，为什么这样使用？

问题一：
首先我们在前面提到，m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent)); 其实是 std::bind( ) 返回了一个函数对象作为 SetEventCallback 的参数，
这用来初始化 data 中的一个元素 EventCallback 。

问题二：那么 BIND_EVENT_FN(OnEvent) 呢？

解释：在定义中我们看到 #define BIND_EVENT_FN(x) std::bind(&Application::x, this, std::placeholders::_1)
意思是为成员函数绑定了对象，并将其作为函数对象传递，这就是前两个参数的意义，
第三个参数：std::placeholders::_1 ， 指出了 OnEvent 的参数暂时被占位了，可以先不填入参数，以便之后处理。

问题三：
之后处理，实际上就是指之后的 data.EventCallback(event); 要进行的处理
通过 Data 类型的对象 data，我们调用出来了刚才初始化进 data 的那个函数：OnEvent。
（我们在之后通过 glfwSetWindowUserPointer 和 glfwGetWindowUserPointer 获取了 m_Data,并将其复制到一个名为data的引用上：WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window); ）
调用出来的 OnEvent( ) 就相当于 data.EventCallback( )，然而 OnEvent 在定义上是需要参数的，所以
data.EventCallback(event) == OnEvent(event) ,这个 event ，就是我们用占位符延缓的参数（这个参数被标明会在后续使用）

在使用 Event 对象作为 OnEvent 的参数填入之后，event这个参数参与到 OnEvent 函数体内的操作中去，完成我们定义的操作。

（在回调函数中我们这样使用：
```
    WindowResizeEvent event(width, height);
    data.EventCallback(event);
)
```

》》》》关于data的使用理解：

```
glfwSetWindowSizeCallback(m_Window, [](GLFWwindow* window, int width, int height)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);
    data.Width = width;
    data.Height = height;

    WindowResizeEvent event(width, height);
    data.EventCallback(event);
});
```

逻辑：WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window); 的作用是从 GLFW 窗口中获取用户指针，并将其转换为 WindowData 类型的引用，从而可以访问窗口相关的数据。
（如果在之前的代码中将 m_Data 设置为窗口的用户指针（通过 glfwSetWindowUserPointer 函数），那么在这个回调函数中获取到的 data 就是之前声明的 m_Data。）

概念：通过 glfwSetWindowUserPointer 来将自定义数据与 GLFW 窗口关联起来，然后在回调函数中使用这些数据。
（这里的 WindowData& data 是对用户指针指向的 WindowData 结构体的 引用（注意：引用），因此对 data 的操作实际上是对窗口关联的数据进行修改或访问。）

深入理解：
（是否反复声明data对象？）
1.每次调用 glfwSetWindowSizeCallback 或 glfwSetWindowCloseCallback 时会重新获取窗口关联的 WindowData 数据，所以不是每次都重新声明 data，
而是获取同一个窗口关联的数据，传入并刷新。
（是否在更新同一个data对象中的值？）
2.是的，多次调用 glfwSetWindowSizeCallback 或 glfwSetWindowCloseCallback 绑定了不同的事件处理逻辑，但是它们都共享同一个 data，每次调用回调时 data 中的值会被修改，因为它们都是指向同一个 WindowData 数据结构的引用。
（是否使用的是私有变量m_Data？）
3.是的，通过 glfwSetWindowUserPointer(m_Window, &m_Data) 将 m_Data 绑定到 GLFW 窗口对象 m_Window 上。而在 glfwSetWindowSizeCallback 的 回调函数中，通过 glfwGetWindowUserPointer(window) 获取绑定在窗口上的 m_Data 结构体的指针，并将其转换为 WindowData& data 引用。
因此，回调函数中的 data 是直接引用并操作了 m_Data 结构体，而不是新声明的结构体。

》》》》为什么在 glfwSetWindowSizeCallback 中要执行 data.EventCallback(event) 这样的操作?

流程：

当窗口大小变化（或是触发某一操作对应的回调函数）时，GLFW 提供的回调函数 glfwSetWindowSizeCallback 会被自动触发，
然后根据我们对该回调函数的定义（定义包含在我们填入的lambda表达式或者函数指针中），在 glfwSetWindowSizeCallback 被自动调用时，
我们会执行到创建相应的事件对象 Event，然后调用之前在 Data 的EventCallback中存入的函数（ EventCallback 所指向的函数 OnEvent），
并将 event 作为参数传递给这个函数。
而这个函数我们是在 m_Window->SetEventCallback(BIND_EVENT_FN(OnEvent)); 这里初始化给Data 的。

效果/目的：

为了确保在特定事件发生时能够调用已经设置好的事件回调函数

》》》》OnEvent为什么要被这样设置？为甚么在每一个回调函数之后都要写一次?

对每一个回调函数实际上都有调用 data.EventCallback(event) ；这用来调用存入data结构体的OnEvent函数，然而在OnEvent中，你可以传入任何事件，
但是只有当事件为WindowClose时候，有对应的处理方式：

dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose)); 这用来实现对窗口关闭时要执行的操作，

但是 HZ_CORE_TRACE("{0}", e); 则是每个回调函数在 data.EventCallback(event); 时都会调用到的语句，是每一个回调函数都能触发的记录的操作。

》》》但是 OnEvent 仅仅只是关于 WindowClose 有对应的操作设计，为什么在 Cherno 的视频中其他的回调函数依旧能正常运行并相应?

1.

```
glfwSetWindowSizeCallback(m_Window, [](GLFWwindow* window, int width, int height)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);
    data.Width = width;
    data.Height = height;

    WindowResizeEvent event(width, height);
    data.EventCallback(event);
});
```

中关键的是

```
data.Width = width;
data.Height = height;
```

data是对M_Data引用，因此Width和Height发生更改时，实际上在glfwCreateWindow这里，窗口就会因为参数的变化而让窗口变化

m_Window = glfwCreateWindow((int)props.Width, (int)props.Height, m_Data.Title.c_str(), nullptr, nullptr);

但关于为什么这里Cherno设置是 props.Width 而不是 m_Data.Width，我不很理解

2.

```
glfwSetWindowCloseCallback(m_Window, [](GLFWwindow* window)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);
    WindowCloseEvent event;
    data.EventCallback(event);
});
```

当然像我们上面所说的，通过 data.EventCallback(event); 进入 OnEvent 中时，

成功满足了 dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose));

故成功关闭

3.

- glfwSetKeyCallback(m_Window, [](GLFWwindow* window, int key, int scancode, int action, int mods)
- glfwSetScrollCallback(m_Window, [](GLFWwindow* window, double xOffset, double yOffset)
- glfwSetCursorPosCallback(m_Window, [](GLFWwindow* window, double xPos, double yPos)

这几个函数，现在并没有去设置按下时应该触发事件，这并不影响当前的操作，

我猜之后可能会在OnEvent中继续续写需要对应执行的一些函数，让事件分发器继续起作用。但现在并没有写。

关于 HZ_CORE_TRACE 能够响应对应键位按下的日志追踪，是因为诸如

- KeyPressedEvent event(key, 0);
- MouseButtonPressedEvent event(button);
- MouseScrolledEvent event((float)xOffset, (float)yOffset);
- MouseMovedEvent event((float)xPos, (float)yPos);

都是从程序自动反复调用回调函数时，从其中的参数中获取了数据，声明了对应的 EventCallback 对象（通过构造函数传入数据）

然后使用 data 的 EventCallback 中存入的 OnEvent，这里有 HZ_CORE_TRACE

OnEvent 接受了这个 Event 对象，然后通过 OnString 成功的获取数据并且打印出来了

（关于为什么会调用了OnString，请看 3game engine 中的这个问题描述：

》》》》WindowResizeEvent WRE(1280, 720);

NUT_TRACE(WRE);

为什么能将WRE作为字符型的类型的参数传入NUT_TRACE这个宏中，并让其中的 trace( ) 函数接受WindowResizeEvent类中ToString函数的结果并输出日志？）

》》》》回调函数的定义结构：理解

```
glfwSetWindowSizeCallback(m_Window,
    [](GLFWwindow* window, int width, int height)
    {
        WindowData& data = *(WindowData*) glfwGetWindowUserPointer(window);
        data.Width = width;
        data.Height = height;

        WindowResizeEvent event(width, height);
        data.EventCallback(event);
    }
);                                  //lambda表达式作为第二个参数
```

使用位置：通常在实际使用中，会在主函数（包括渲染循环）之前，初始化GLFW窗口之后进行回调函数的定义。

接受参数：一般回调函数接受两个参数，1.窗口对象，2.一个函数。

逻辑流程：在特定的事件比如窗口大小发生变化时候，回调函数会自动的监测到操作，并且获取数据。

（具体的说，应该是 GLFW 负责传递相应的数据给回调函数，然后回调函数再将这些数据传递给用户定义的处理函数。）

随后便会自动的去调用用户自己传入的函数，因为回调函数本身并不会进行任何操作，这些都需要用户自己定义。

提示：这个函数可以是在某处定义的，然后传入这个函数指针，或者也可以是一个lambda表达式（在作为参数的时候，可以就地定义的函数）

》》》》对于一个数据 double pos，使用 (float)pos 和 float(pos) 这两种方式的类型转换有什么不同

1.(float)pos 是一种 C 风格的类型转换方式。这种方式在 C++ 中仍然有效，但不够安全，因为它可以进行任意类型的转换，包括隐式转换和强制转换，可能会导致潜在的错误。
2.float(pos) 是一种 C++ 风格的类型转换方式，称为函数风格的强制类型转换（functional cast）。这种方式在 C++ 中更为推荐，因为它提供了更明确的类型转换操作，同时在某些情况下还能提供更好的类型安全性。
（会有警告但不会影响正常运行）

》》》》关于事件分发器Dispatcher的语法分析：
1. using EventFn = std::function<bool(T&)>;
声明了 EventFn 作为一个类型，这个类型代表一个接受 T 类型参数并返回 Bool 类型的函数，所以在我们后续使用时候，填入的 WindowCloseEvent

2.
```
EventDispatcher(Event& event)
             :m_Event(event)  {}
```

首先初始化一个对象

3,
```
template<typename T>
  bool Dispatch(EventFn<T> func) {                              // (这里将会在未来使用中填入一个函数指针)
      if (m_Event.GetEventType() == T::GetStaticType()) {       //!!! 静态函数在使用时需要使用类名或类型名来调用 (T::)
          m_Event.m_Handled = func(*(T*)&m_Event);              //*(T*) 表示：用 * 解引用 (T*) 所声明的T类型指针，实现强制类型转换
          return true;  }
      return false;
  }
```

后续我们会这样使用：

```
Event& e;         //作为  OnEvent  的一个参数传入的
EventDispatcher dispatcher(e);
dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose));
```

思路：
首先用 e 初始化了一个对象，然后通过 dispatcher 的成员函数 Dispatch 将一个接收 T 类型的返回 bool 类的函数作为对象填入
（这个函数是 OnWindowClose ，只不过用 std::bind 将其作为函数对象传入，因为我们将延迟一些操作，所以不使用 std::functional）

然后，我们标明了 template<typename T> 为 WindowCloseEvent ，于是我们进行判断：
如果通过 OnEvent 传入的 Event e 与我们标明的 typename T 一致，
则说明这个事件与我们想要允许调用 BIND_EVENT_FN(OnWindowClose) 的事件是一致的
这就是：if (m_Event.GetEventType() == T::GetStaticType()) 的作用。

进入条件判断语句内部之后，m_Handled 被赋予了我们填入的函数的返回值，否则返回 false，即不允许该事件类型执行此函数

》》那么那一句是他执行我们定义的函数的语句呢？

首先我们 dispatcher.Dispatch<WindowCloseEvent>(BIND_EVENT_FN(OnWindowClose));
将函数传入，但是通过 std::bind 我们使用一个占位符延缓了参数填入的时间，在 Dispatch 中，
在 if (m_Event.GetEventType() == T::GetStaticType()) 的条件下使用了 m_Event.m_Handled = func(*(T*)&m_Event);
func(*(T*)&m_Event) 便是对填入的自定义函数的调用，同时将 m_Event 强制转换为允许的类型，在此处调用函数

》那为什么在赋值的过程中还可以进行函数中的操作呢？？
流程：
1.当调用一个函数时，函数体内的语句会被按顺序执行。
2.函数可以有返回语句或者没有返回语句。如果没有返回语句，则函数会自动返回一个默认值（对于 bool 类型，未显式返回的函数会返回 false）。
3.当函数执行完所有语句后，会将返回值传递给调用者。

----------layers----------

》》》》LayerStack.cpp中的语句大致意思：
LayerStack::LayerStack()：
　　初始化了 m_LayerInsert，它是一个迭代器，用于指示下一个被插入图层的位置。初始时，它指向 m_Layers 的开始位置。
LayerStack::~LayerStack()：
　　释放所有图层的内存。它遍历 m_Layers 并使用 delete 删除每个图层的指针，确保不会发生内存泄漏。

LayerStack::PushLayer(Layer* layer)：
　　将一个 “普通图层” 推入图层列表的前半部分。使用 m_Layers.emplace 在 m_LayerInsert 位置之前插入新图层，然后更新 m_LayerInsert 指向新的位置。
LayerStack::PushOverlay(Layer* overlay)：
　　将一个 “覆盖图层” 推入图层列表的后半部分。使用 m_Layers.emplace_back 在 m_Layers 的末尾添加新图层。

LayerStack::PopLayer(Layer* layer) 和
LayerStack::PopOverlay(Layer* overlay)：
　　从图层列表中删除指定的图层。它们使用 std::find 在 m_Layers 中找到要删除的图层，并使用 erase 函数从列表中移除。在删除图层后，m_LayerInsert 更新为指向正确的位置。

》》》》关于“普通图层”和“覆盖图层”的理解
在CS:GO游戏中
普通图层：一般是游戏场景和玩家角色，它们包含了游戏世界的内容以及玩家的交互。
覆盖图层：一般是设置菜单、商店界面等UI，它们会覆盖在游戏场景上方，用于显示各种菜单、选项、提示等用户界面元素。

》》》》层的传入顺序

层的设置：

层这个数组整体分为两部分：前半部分为普通图层，后半部分为覆盖图层。

就绪后从头到尾开始绘制。（layer3, layer2, layer1, overlay1, overlay2, overlay3）

层的就绪：

上述的函数：

```
void LayerStack::PushLayer(Layer* layer) {
    m_LayerInsert = m_Layers.emplace(m_LayerInsert, layer);
    //此操作不仅将 layer 插入到指定位置，还会将返回的迭代器赋值给 m_LayerInsert，
    以便下次插入时插入在这个位置，并将此前这个位置的元素后移一位。 }

void LayerStack::PushOverlay(Layer* overlay) {
    m_Layers.emplace_back(overlay); }
```

解释与结果：

调用 PushLayer 推入普通图层到数组前半部分时，新图层会被插入到 m_LayerInsert 迭代器所指向的位置之前。因此，如果 m_LayerInsert 最初指向 layer1，那么插入 layer1 应该是这样的顺序：layer3, layer2, layer1。

调用 PushOverlay 函数将覆盖图层推入图层列表的后半部分时，新图层会被添加到列表的末尾。因此，使用overlay1, overlay2, overlay3 分别表示推入的覆盖图层应该是这样的顺序：overlay1, overlay2, overlay3。

》》》运行流程（以CSGO举例的话）

1.图层将会是

（layer3, layer2, layer1, overlay1, overlay2, overlay3）

其中普通图层先后由 layer1：游戏UI、layer2：游戏角色、layer3：游戏背景

覆盖图层先后由 overlayer1：菜单、overlayer2：设置、overlayer3：设置中的一个选项

层栈结构：

overlay3　某一设置选项图层
overlay2　设置图层
overlay1　菜单图层
layer1 游戏中的UI（击杀敌人时显示的图标）
layer2 游戏角色
layer3 游戏背景

2.事件将会是：

从最后的图层开始处理。（要对最顶层的界面进行交互，退出后才能对低一层的界面操作，否则不符合直觉和视觉。）

所以以一直在进行交互的这个图层永远是最顶层的图层。

》》拓展：

（即便是在一个图层上新生成一个图层，也是由此前作为最顶层的图层做处理，由他生成一个图层，
然后该屈居较低一层，使新图层作为最顶层，以便进行操作）

》》事件反向处理的原因：

1.符合人的操作习惯：先处理最上层的图层可以更快地响应用户的操作。

2.符合人的视觉习惯：保持游戏画面的逻辑性，避免混乱和不连贯的情况出现。

》》》》层栈结构的理解

层栈结构：

overlay3　某一设置选项图层
overlay2　设置图层
overlay1　菜单图层
layer1 游戏中的UI（击杀敌人时显示的图标）
layer2 游戏角色
layer3 游戏背景

问题发现与分析：

1.层栈只是一个理想上的栈结构，实际上Cherno只设置了一个 vector 来存储图层，所以这个图层结构是
layer3, layer2, layer1, overlay1, overlay2, overlay3 躺地上的。

！！！！

2.因为在 pushlayer 函数中我发现emplace函数会将元素插入到当前位置之前，
所以我一直在考虑传入普通图层的时候是否要特定顺序，比如先传入较高层次的图层？
但是覆盖图层却是先传入较低层次的图层（这和两个push函数传入元素的方向有关系）

2.另外我注意到Cherno所说，“我们将覆盖图层放在列表最后，我们总是希望他最后渲染”，所以我猜测在这里，Cherno决定的渲染顺序是
直接从栈底（vector头部）开始向上（vector尾部）处理

最终总结：

1.事件处理：从栈顶到栈底（从vector尾部到头部）

2.渲染处理：

第一种（当普通图层先传入较高层次，覆盖图层先传入较低层次情况下）

overlay3　某一设置选项图层
overlay2　设置图层
overlay1　菜单图层　↑
layer1 游戏中的UI（击杀敌人时显示的图标）　↓
layer2 游戏角色
layer3 游戏背景

渲染方向：从栈底到栈顶（从vector头部到尾部）

第二种（当普通图层和覆盖图层都是先传入较低层次时）

overlay3　某一设置选项图层
overlay2　设置图层
overlay1　菜单图层　↑

layer1 游戏背景 ↓
layer2 游戏角色
layer3 游戏中的UI（击杀敌人时显示的图标）
渲染方向：普通图层部分从顶到底，之后覆盖图层从底到顶

不知道我的分析是否正确，但以此看来两种方法各有裨益。
还是要根据Cherno后续的操作来分析

》》》》emplace函数实际情况
1.循环通过 emplace 向 vector 中传入元素时，会将新元素放在先前元素之前，并且返回一个指向最新的元素的指针

layer3 layer2 layer1
↑
指针

2.越晚传入的元素地址越大

layer3 layer2 layer1
指针地址：3 2 1

》》》》pop函数中Insert--的作用？？？？？？？？，判断条件可以用来删除除栈顶元素以外的其他元素吗
在我反复观看代码，并且理解层栈结构之后，我认为：
前提：
在推入三个layer之后，Insert是在Layer3这里指着的。
1.递减的设计思考：
虽然 Insert-- 确实会将指针指向下一个元素，但是这完全建立在删除的元素一定是最晚传入（栈顶元素）的基础上。
如果删除栈顶元素， Insert 由栈顶被移到下一个元素上，并且这下一个元素接替栈顶的位置。
否则 Insert-- 在删除其他元素时是完全没有其他意义的。
2.判断条件的思考：
2.那既然已经这样设计了，那Insert--就只能在删除固定的、明确的、栈顶元素情况下使用了
所以，这前面的判断条件便仅仅为了确保所要删除的元素存在 vector 中
并不存在用 std::find 去寻找除栈顶以外的元素并将其删除的思路了。

结论：
在层栈中，PopLayer函数用于删除普通图层这一部分的栈顶，
1.Insert--的作用是将指向头部的指针位置移向下一个充当栈顶的元素这里
2.判断条件式确保删除操作的安全性，避免删除其他数据。并无其他作用。

对于PopOverlay来讲，并没有提供 Insert 这一指针（实际上是迭代器类型，可以这样理解）
所以只需传入 overlay 的栈顶元素，直接调用erase即可。

》》》》erase 函数的参数及其用法
1.erase(iterator pos)：
删除指定位置的元素。pos是一个指向待删除元素的迭代器。
2.erase(iterator first, iterator last)：
删除指定范围内的元素。first和last是表示范围的迭代器，删除的元素包括first指向的元素，但不包括last指向的元素。

》》》》layer stack中的
std::vector<Layer*>::iterator begin() { return m_Layers.begin(); }
std::vector<Layer*>::iterator end() { return m_Layers.end(); }
为什么需要这两个函数？

1.首先要了解基于范围的 for 循环这个概念：
概念：基于范围的 for 循环也称 for each 循环，是一种简化遍历容器元素的语法。
使用要则：允许使用者直接遍历容器中的元素，而不必使用迭代器和循环索引。
2.现在了解这两个函数的作用；
因为在 for each 循环中，会使用容器确定整个 for each 循环的范围，这就要求使用的容器具有 begin() 和 end() 成员函数来返回迭代器。
以便正确遍历容器中的元素。

》》》》一个错误分析
```
for (auto it = m_LayerStack.end(); it != m_LayerStack.begin(); )
{
    (*--it)->OnEvent(e);
    if (e.Handled)
        break;
}
```

这里的 (*--it) 可以写为 *(--it) 吗？
理论上来讲，这二者并没有区别，但是在实际使用中可能会由于优先级的问题发生问题。
所以要么 (*--it)->OnEvent(e); 要么 (*(--it))->OnEvent(e);

》》》》sandbox 中的函数和 项目Nut (Hazel) 有什么关系，这些函数是怎样能够影响到 application 中的函数的？
他们是怎样传递的？

1.在代码中，Sandbox 类的构造函数
```
Sandbox() {
    PushLayer(new ExampleLayer());
```

}

通过 PushLayer ( ) 创建 ExampleLayer 图层对象并将其添加到 LayerStack 中，
（而且 PushLayer 函数是 Nut 项目中 LayerStack 类中的一个函数）

2.在项目 Sandbox 中，ExampleLayer 这个类继承自 Nut::Layer，并且重写了 OnUpdate 和 OnEvent 函数。
这意味着当你创建 ExampleLayer 对象并将其添加到 LayerStack 中后，这些重写的函数会在你对 ExampleLayer 对象进行操作时被"对应的"调用
（因为在 ExampleLayer 中，这几个函数被重写了，并且作为 ExampleLayer 这个类的成员函数）

比如在 application.cpp 中，(*--iter)->OnEvent(e); 就是自动辨别 iter 的类型，然后自动的使用了这个类下的成员函数 OnEvent

- ---

所以，在 Application 类的 Run 函数中，你遍历 m_LayerStack 并调用其每个图层的 OnUpdate 函数。
但是由于 ExampleLayer 是 Layer 的子类，所以当你对刚才传入的 LayerStack 的 ExampleLayer 类型的对象进行操作时，ExampleLayer 中重写的 OnUpdate 函数会被调用。

- ---

同样，在 Application 类的 OnEvent 函数中，你也遍历 m_LayerStack 并调用每个图层的 OnEvent 函数。
由于 ExampleLayer 也重写了 OnEvent 函数，所以在这里也会调用 ExampleLayer 中重写的 OnEvent 函数。

----------OpenGL & Glad----------

》》》》int status = gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);是在干嘛?
作用：
使用 glfwGetProcAddress 获取当前环境下的 OpenGL 函数的地址，并通过 gladLoadGLLoader 将这些函数指针加载到程序中，从而使得程序可以调用 OpenGL 提供的各种函数进行图形渲染等操作。

之前都在使用glfw，经此之后可以使用gl的函数，以此获取用来进行图形渲染的函数。

》》》》一个不同：
Cherno在程序中为了避免 glad.h 和 glfw3.h 包含两个 gl.h（好像是这个问题？）导致的错误
选择加入一个宏定义以确保 gl.h 不会被包含两次。

但是我记得在之前学习图形渲染的时候，有一个方法也行，就是先包含 glad.h：
#include <glad/glad.h>
#include <GLFW/glfw3.h>
于是我在WindowsWindow.h中按照这样的方式同时包含两个文件，并在 WindowsWindow.cpp 中删除 glad.h。这样也是可以正常运行的。
参考：（https://learnopengl-cn.github.io/01%20Getting%20started/03%20Hello%20Window/）

----------ImGui----------

》》》》因为真的很想复刻Cherno的操作（而且对于最新版本的imgui我也不是很理解其文件构架）
所以我决定将Cherno当时使用的imgui版本作为一个库，然后使用。
（ http://t.csdnimg.cn/67Snv ）
这是我的方法，如果需要可以查看

》》》》什么是单例模式?
概念：
单例模式是一种设计模式，用于确保类只有一个实例，并提供一个全局访问点来访问该实例。

在以下情况下，可以考虑使用单例模式：
1.全局访问点：
当需要在整个应用程序中共享某个对象实例时，单例模式可以提供一个全局访问点，使得任何地方都可以方便地获取到这个实例。
2.资源共享：
在需要共享资源的情况下，比如数据库连接池、日志文件等，可以使用单例模式确保资源的唯一性和合理的管理。
3.控制实例个数：
有些情况下，系统中某个类只能有一个实例，比如线程池、缓存、配置文件等，这时可以使用单例模式来限制实例个数。
4.节省内存：
有些对象占用大量内存，频繁创建销毁会带来性能问题，使用单例模式可以避免重复创建实例，节省内存空间。
5.全局状态管理：
在需要维护全局状态的场景下，比如全局配置信息、用户登录信息等，单例模式可以提供一个统一的状态访问接口。

》》》》单例模式的讲解：
（https://www.bilibili.com/video/BV1bR4y177Hp/?spm_id_from=333.999.0.0&vd_source=64ca0934a8f5ef66a21e8d0bddd35f63）

为什么在代码这里需要使用单例模式呢?
因为我们在游戏引擎中只会需要一个Applicaiton，所以我们使用了单例模式。

----------ImGui事件----------

》》》》鼠标事件函数为什么最后一句是return false;事件消费是什么?
为什么 ImGuiLayer::OnMouseButtonPressedEvent 最后需要返回 false ?

当处理事件时，返回 false 通常表示事件没有被“消费”，即并未完全处理。在这种情况下，我们可能希望其他地方也有机会继续处理这个事件。如果函数返回 true，则表示事件已经被处理了，不需要进一步传播。
因此，在这段代码中，返回 false 是为了让其他地方也有机会处理鼠标按键按下事件。

举例：
eg1:
有时候，我们会选择长按某一按键达成某种操作。这就要求我们在一帧结束之后，继续询问并处理该事件。
否则想达成长按按键时，却会因为事件被提前“消费”而中断操作。
eg2:
在打开的商城页面中，在点下商品之后，此图层并不会消失，而是等待另一个按钮“购买”被点击后该图层才会消失。
1.点击商品时，整体事件并没有处理完
2.为了购买按钮的事件能够触发，我们将前一个事件标记为未处理完成，将其进一步传播
3,直到购买按钮被触发，整个事件完成

》》》》glViewport的参数

概念： 设置视口（Viewport），用来指定 OpenGL 渲染的目标区域在帧缓冲区中的位置和大小。
原型： void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
参数：
x：指定视口的左下角 X 坐标。
y：指定视口的左下角 Y 坐标。
width：指定视口的宽度。
height：指定视口的高度。

》》》》io.KeyCtrl = io.KeysDown[GLFW_KEY_RIGHT_CONTROL] || io.KeysDown[GLFW_KEY_LEFT_CONTROL];
的逻辑是什么？

如果任一 Ctrl 键被按下，则 io.KeysDown[GLFW_KEY_？？_CONTROL] 的值为 true，否则为 false。
左右 Ctrl 键的按下状态进行逻辑或运算，最终将结果赋值给 io.KeyCtrl ，表示用户是否按下了任意一个 Ctrl 键。

》》》》关于WindowsWindow.cpp中的回调函数和ImGuiLayer.cpp中的OnKeyTypedEvent的关系
为什么这两个函数有所联系？？？

```
glfwSetCharCallback( m_Window, [](GLFWwindow* window, unsigned int keycode)
    {
      WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);

      KeyTypedEvent event(keycode);
      data.EventCallback(event);
    }
);

bool ImGuiLayer::OnKeyTypedEvent(KeyTypedEvent& e)
{
    ImGuiIO& io = ImGui::GetIO();
    int keycode = e.GetKeyCode();
    if (keycode > 0 && keycode < 0x10000)
        io.AddInputCharacter((unsigned short)keycode);

    return false;
}
```

因为在回调函数glfeSetCharCallback中有通过回调函数获取的keycode构建KeyTypedEvent对象的过程 -->即 KeyTypedEvent event(keycode);
此时我们就获取了键盘上输入的关键字，而且存入了这个事件对象中，然后在之后的ImGuiLayer上的demo窗口中，我们使用int keycode = e.GetKeyCode();获取了Keycode，
所以相当于我们从回调函数中获取了keycode这个数据，然后在demo窗口中我们就能够使用它。

----------Github & Repo----------
》》》》

----------Pull requests----------

》》》》startproject "Sandbox"，为什么 Error: unable to set startproject in project scope, should be workspace
startproject 命令应该在工作区的作用域内. 要放置在 Premake 脚本的顶层.

```
eg:
workspace "Work_space"
    configurations { "Debug", "Release" }
    ......

    startproject "Sandbox"  -- 将启动项目设置为 "Sandbox"

project "Sandbox"
    kind "ConsoleApp"
    language "C++"
    ......
```

----------polling input轮询输入----------

》》》》独立窗口
概念：
一个在操作系统中独立存在的、可以单独打开、关闭和移动的窗口。
特点：
可以被拖动到屏幕上的任何位置
可以被最小化、最大化和关闭
可以独立于其他窗口存在

》》》》静态类

概念：

一个类

1.只包含静态成员（静态属性、静态方法等）

2.不能被实例化

3.也不能继承其他类

用途：

作为工具类：类中包含一组静态方法，二这些方法能够提供一些通用的功能（计算、日期、操作...）

在使用这些方法时，我们不需要实例化该类就能使用其中的静态成员函数。

全局访问点：静态类中的静态成员可以被全局访问，以供整个程序使用这些方法或者属性。

常量集合： 静态类可以用于存储常量，在需要时可以通过类名直接进行访问

举例：

单例模式就是一种静态类的例子

》》》》在GetMousePosImpl中，返回了{ sth_about_std::pair<> }

而在GetMouseX中，为何使用auto [x,y] 来接受参数?

原因：

这其中涉及到了结构化绑定。

什么是结构化绑定?

概念:

结构化绑定（Structured Binding）是 C++17 中引入的一个特性，

它允许将一个复合类型（如 pair、tuple、数组等）的成员解构为单独的变量。

语法：

auto [var1, var2, ...] = expression;

参数：

var1, var2, ... 是要绑定的变量名，用逗号分隔。

expression（表达式） 是返回复合类型的表达式，可以是函数返回值或其他包含多个值的表达式。

作用：

这样可以方便地从复合类型中提取各个成员，并将它们赋值给单独的变量。

要求/规范：

（只能对以下类型的对象使用）：

标准库中的 tuple 类型：可以将 tuple 的各个元素解构为单独的变量。

标准库中的 pair 类型：可以将 pair 的两个成员解构为单独的变量。

数组：可以将数组的各个元素解构为单独的变量。

eg.

在 auto [x, y] = GetMousePosImpl(); 中，GetMousePosImpl() 返回一个 std::pair<float, float> 类型的对象

而通过结构化绑定 auto [x, y]，这个对象被解构为两个单独的变量 x 和 y，可以被单独使用。

----------Keycodes & MouseButtonCodes（键盘和鼠标代码）----------

》》》》

没什么要记的

----------数学库----------

》》》》glm/glm/gtc 和 glm/glm/ext 的区别

注意到 github 上最新的例子引用的头文件是来自 ext 中的头文件，而glm 9.9.0 版本示例使用了 gtc 中的文件，故发问。

gtc：

包含一组便利性函数和工具，用于扩展 GLM 的功能（矩阵变换、投影等）提供许多常用的数学运算和变换函数，方便开发者快速实现各种数学操作。

ext:

包含了 GLM 的扩展功能（一些实验性质的函数或者功能更为专业化的内容）可能还在开发中或者不太稳定。