2.概念与操作

2024年5月24日 15:38

》》》》写在前面

win + x	打开快速访问菜单（包含一系列常用的系统管理工具和设置选项）
win + i	打开设置
Ctrl+shift+a	打开创建文件的页面
ctrl +j （alt + → ）	（在VS中）可以强制弹出当前对应的补全内容，这样就不用每次都删了重写来显示补全的信息了，很常用的一个小功能
Ctrl + Tab：	（VS中）按住Ctrl键并连续按下Tab键，可以在打开的文件之间进行切换。
Alt + D：	浏览器窗口中的地址栏会被选中并高亮显示，这使您可以立即开始输入网址或搜索内容
Win + R，输入Notepad：	打开记事本
.LOG：	为记事本使用时间戳
Shift + Ctrl + N：	新建文件夹
..\ ：	表示上一级文件
##	是预处理操作符，用于将宏参数连接起来。（EventType::##type 就会被展开为 EventType::Mouse）
#	是字符串化操作符，将宏参数转换为字符串。return #type; 如果 type 是 Mouse，那么 #type 就会被展开为 "Mouse"。
ctrl + n:	创建一个新的文件资源管理器界面（在文件资源管理器界面）
ctrl + L:	选中当前行（ VS2019 中好像被设置为剪切：参考官网）
Ctrl + Esc:	打开菜单（相当与单击 win ）
Ctrl + E:	和 Alt + D 类似（Ctrl + E 适合在进行搜索的时候快速定位到搜索框，而 Alt + D 更适合在直接编辑网址的时候快速定位到地址栏。）
Win + A:	打开快速设置页面
Ctrl + Shift + Space:	（在VS中）打开函数参数的提示框

VS2019中打开 minimap:	工具->选项->文本编辑器->C++->使用垂直滚动条的缩略图模式	(https://learn.microsoft.com/zh-cn/visualstudio/ide/how-to-track-your-code-by-customizing-the-scrollbar?view=vs-2022)

Win+Shift+N:	打开快速笔记 OneNote，或者在 Win+R 中输入 onenote 指令。
Win + Alt + S:	打开便签（小型的笔记本，用于临时记载一些东西）
Ctrl + Alt + L:	在VS2019中打开解决方案资源管理器的快捷键
Ctrl + Q :	在VS/One note中，为打开搜索框。但在浏览器中好像没有实际作用，
Ctrl + K, Ctrl + D	（在VS中）格式化整个文件的代码
Ctrl + K, Ctrl + F	（在VS中）格式化选中的代码区域
Ctrl + B	（在VS中）好像是当前项目重新生成的快捷键
Shift + F10	相当于鼠标的右键
Ctrl + C	（在控制台中）暂停当前在运行的进程
Ctrl + /	（VsCode中）代码快捷注释
Ctrl + K, Ctrl + C	（在VS中）快捷注释代码
Ctrl + K, Ctrl + U	（在VS中）快捷解除注释

》》》》预定义的宏

参考网站：（ https://learn.microsoft.com/zh-cn/cpp/preprocessor/predefined-macros?view=msvc-170 ）

以下列举一些常用的：

标准预定义宏

`__DATE__`：	当前源文件的编译日期。 日期是 Mmm dd yyyy 格式的恒定长度字符串文本。 月份名 Mmm 与 C 运行时库 (CRT) asctime 函数生成的缩写月份名相同。 如果值小于 10，则日期 dd 的第一个字符为空格。 任何情况下都会定义此宏。
`__FILE__`：	当前源文件的名称。 `__FILE__` 展开为字符型字符串文本。 要确保显示文件的完整路径，请使用 （诊断中源代码文件的完整路径）/FC。 任何情况下都会定义此宏。
`__LINE__`：	定义为当前源文件中的整数行号。 可使用 `#line` 指令来更改此宏的值。 `__LINE__` 值的整型类型因上下文而异。 任何情况下都会定义此宏。
`__TIME__`：	预处理翻译单元的翻译时间。 时间是 hh:mm:ss 格式的字符型字符串文本，与 CRT asctime 函数返回的时间相同。 任何情况下都会定义此宏。

Microsoft 专用预定义宏

`__COUNTER__`：	展开为从 0 开始的整数文本。 每次在源文件或源文件的 `included` 头中使用时，此值都会递增 1。 当使用预编译头时，`__COUNTER__` 会记住其状态。 任何情况下都会定义此宏。
`_CPPUNWIND`：	如果设置了一个或多个 /GX （启用异常处理）、/clr （公共语言运行时编译）或/EH （异常处理模型）编译器选项，则定义为 1。 其他情况下则不定义。
`_DEBUG`：	当设置了 /LDd、/MDd 或 /MTd 编译器选项时，定义为 1。 其他情况下则不定义。
`_DLL`：	当设置了 /MD 或 /MDd（多线程 DLL）编译器选项时，定义为 1。 其他情况下则不定义。
`__FUNCDNAME__`：	定义为包含封闭函数修饰名的字符串文本。 此宏仅在函数中定义。 如果使用 /EP 或 /P 编译器选项，则不会展开 `__FUNCDNAME__` 宏。

》》》》编译和生成的区别：

- 编译: 将源代码（如 C++、Java 等）转换为中间代码或机器代码。这是编译过程的核心，确保代码的语法和语义正确。
- 生成: 包括编译和链接过程。生成不仅涉及将源代码编译成目标文件，还包括将这些目标文件和其他资源（如库）链接成最终的可执行文件或库。

》》》》“x64(active) 平台”和“x64 平台”的区别：

x64(active) 平台：在Visual Studio中，"x64(active)" 是指针对 64 位处理器架构的项目配置。选择这个配置意味着项目将会被编译成针对 64 位处理器的可执行文件，可以充分利用 64 位处理器的性能和内存寻址能力。

x64 平台： 同样也是指针对 64 位处理器架构的项目配置。在较早的版本的Visual Studio中，可能会看到简单的 "x64" 选项，它与 "x64(active)" 实质上是相同的，都代表了针对 64 位处理器的项目配置。

在新版的Visual Studio中，可能会看到 "x64(active)" 选项，这是为了更清晰地表示当前项目配置是针对 64 位处理器的。因此，在这种情况下，“x64(active) 平台”和“x64 平台”其实是指同一个概念，只是一种是较早版本的表示方式，另一种是较新版本的表示方式。

》》》》宏（条件判断的实现逻辑）

宏不会自动定义。如果在 属性页 -> C++ -> 预处理器 中填入一个宏XXX，
这意味着在编译代码时会自动在预处理阶段为XXX这个宏定义一个值。

这样不用手动编写一个宏，可以直接使用#ifdef语句进行条件判断。

```
<---- Eg.

#define 宏名称 值或代码

#ifdef 标识符
    // 如果标识符已经被定义，则编译这部分代码
#else
    // 如果标识符没有被定义，则编译这部分代码
#endif
```

》》》》什么是API，OpenGL是API吗？GLFW是API吗？

1.API（Application Programming Interface，应用程序编程接口）是一组定义了软件组件如何相互交互的规范。
它定义了一组规范、协议和工具，允许不同的软件组件（如库、框架、操作系统等）之间进行通信和交互。

2.OpenGL 是一个图形渲染 API，它定义了一系列函数和数据结构，用于执行各种图形操作。
OpenGL 提供了一个标准化的接口，使得开发人员可以在不同的平台上编写图形应用程序，而无需关心底层硬件的细节。
因此，OpenGL 是一个 API。

3.GLFW（Graphics Library Framework）是一个用于创建窗口和处理用户输入的库，它并不是一个 API，而是一个库（Library）。
GLFW 提供了一系列函数辅助开发者创建基于 OpenGL 的图形应用程序，它使用 OpenGL 的 API 来实现其功能。

》》》》什么是打包？

封装（Packaging）可以理解为打包的一种表达方式，在软件开发中常用来指代将相关文件和资源封装到一个独立的包中以便于分发和使用。
（打包和封装在大部分情况下可以视作同义词。）

打包（Packaging）指的是将软件或应用程序的源代码、依赖项和其他必要的资源组合在一起，以便在其他环境中进行部署或分发。
它可以将一个或多个文件或目录打包成一个单独的可执行文件、库文件、安装程序、容器镜像等形式。

打包通常包括以下内容：

源代码：包括软件的原始代码文件，用于编译和构建可执行文件或库。
依赖项：软件所依赖的库文件、框架、工具或其他第三方组件。这些依赖项可能需要事先安装或打包到同一个包中，以确保软件在目标环境中能够正常运行。
资源文件：例如配置文件、模板、静态文件、图像、文档等，这些文件通常是软件运行所需的辅助资源。

```
eg:
class Calculator {
public:
    int func( int a ){
        return XXX::complex_function( a );
    }
};

int main(){
    Calulator calc;
    std::cout << "result is " << calc.func( X );
}
```

》》》》共享指针（智能指针）

特点：

1.引用计数：共享指针使用引用计数来追踪有多少个指针共同拥有一个对象。并自动管理对象的生命周期。

case 1：	每当创建一个共享指针指向对象时，引用计数加一；
case 2：	当销毁一个共享指针或者将其重新分配给其他对象时，引用计数减一。
case 3：	当引用计数为零时，表示没有指针指向对象，对象会被自动销毁。

2.多个指针共享所有权：共享指针允许多个指针同时拥有对同一个对象的所有权，这使得多个部分可以共享对象的状态和数据。

eg:
使用 std::shared_ptr 创建两个共享指针 ptr1 和 ptr2，它们都指向同一个 MyClass 对象。
通过共享指针，我们可以同时使用 ptr1 和 ptr2 访问和管理对象。

》》》》静态函数

静态函数是属于类而不是对象的函数，它们没有隐式的 this 指针，并且可以直接通过类名进行调用。
（静态函数适用于执行与类相关但不依赖于类的实例的操作，共享资源或封装辅助函数的场景。）
特点：

1.在一个静态成员函数中，只能访问静态声明的成员变量。
2.对于静态成员变量，你必须在某个地方进行定义。（并且定义处可以选择是否进行初始化）
这是因为静态成员变量属于类而不属于类的对象，所以需要在类外进行定义。
如果你没有提供定义，编译器将无法找到静态成员变量的实际存储位置，从而导致链接错误。

```
eg:
class log.h:
    class NUT_API Log
    {
    private:
        static std::shared_ptr<spdlog::logger> s_CoreLogger;
    public:
        inline static std::shared_ptr<spdlog::logger>& GetCoreLogger
    };

log.cpp:
std::shared_ptr <spdlog::logger> Log::s_CoreLogger;
```

》》》》::namespace::func();
"::Nut" 中的 "::" 表示全局命名空间，即根命名空间。表示访问全局命名空间中的 Nut 命名空间。

```
eg:
namespace Nut {
    void foo() {}
}

int main() {
    ::Nut::foo();       // 调用全局命名空间中的 Nut 命名空间下的 foo 函
    return 0;
}
```

》》》》双下划线 "__" ----> 预定义的宏
定义： 双下划线 "" 表示这是一个预定义的宏，由编译器或标准库定义。
目的： 一些预定义的宏都包含双下划线 "__"，例如 __cplusplus、LINE、FILE 等等。这样设计的目的是为了避免与用户自定义的标识符冲突，并且提供一些方便的功能。

》》》》(...) 和 __VA_ARGS__配对使用

1.(...) 是可变参数模板的语法，表示宏函数可以接受任意数量的参数。
2.VA_ARGS 是一个预定义的宏， 在 C++ 中用于表示可变参数列表。它将被展开成实际传入的可变参数列表。

一般情况下，在宏定义中使用 (...) 来接受可变数量的参数，在宏展开时使用 VA_ARGS 来引用这些参数。
下面是一个示例来说明 (...) 和 VA_ARGS 的配对使用：

```
#define PRINT_VALUES(format, ...) \
    printf(format, __VA_ARGS__);

int main() {
    PRINT_VALUES("%d %s\n", 10, "Hello");  // 输出：10 Hello
    return 0;
}

```

在这个例子中，PRINT_VALUES 宏使用了可变参数模板 (...) 来接受可变数量的参数，然后使用 VA_ARGS 来引用这些参数。
在宏展开时，VA_ARGS 将被实际传入的可变参数替换。

》》》》什么是bat文件
概念： BAT 文件是批处理文件的一种，它是一种包含一系列命令的文本文件，用于在 Windows 操作系统中批量执行命令。批处理文件使用扩展名为 .bat 或 .cmd。
用处： BAT 文件可以包含命令行命令、控制流语句（如条件判断和循环）、变量定义和其他批处理脚本语言的特性。
在BAT文件中输入call commands_in_cmd可以实现启动此BAT文件时，在当前路径的命令行中执行该命令。

》》》》什么是premake，什么是Lua
概念： Premake 使用 Lua 作为其脚本语言。
Lua 是一种轻量级、高效、可嵌入的脚本语言，非常适合用于配置文件和脚本语言的编写。

详见官方文档：（https://premake.github.io/docs/）

eg: 以下是 Premake 中 Lua 的一些基本语法和使用方法：

```
//1.注释：使用双连字符 "--" 进行单行注释，或者使用长注释形式 "--[[ ...
    --  单行注释
    --[[
        这是一个
        多行注释
    ]]


//2.变量：无需声明变量类型，直接赋值即可。
    name = "John"
    age = 25

//3.表（Table）：类似于字典或哈希表的数据结构，可以存储键值对。
    person = {
      name = "John",
      age = 25,
      gender = "male"
    }

//4.函数，使用 function 关键字定义函数。
    function greet(name)
      print("Hello, " .. name .. "!")
    end

    greet("John") -- 输出：Hello, John!

//5.控制流，支持条件语句（if-else）和循环语句（for、while）。
    if condition then
      -- 条件为真时执行的代码
    else
      -- 条件为假时执行的代码
    end

    for i = 1, 5 do
      print(i)
    end

    while condition do
      -- the 循环体
    end

//6.Premake API：Premake 提供了一组 API 来定义项目的属性、构建规则和配
-- 定义项目
project "MyProject"
  kind "ConsoleApp"
  language "C++"
  files { "src/*.cpp", "include/*.h" }
  includedirs { "include" }
  links { "LibraryA", "LibraryB" }

-- 定义构建规则
filter "system:windows"
  defines { "WINDOWS" }

```

》》》》友元类和派生类的区别：
1.友元类是一种权限控制机制， 用于让一个类可以访问另一个类的私有和受保护成员，而不是继承关系。
2.派生类是一种继承关系， 用于创建一个新的类并从另一个类继承属性和方法，可以对继承的成员进行扩展和修改。{重写（override）}
派生类可以访问基类中的公有和受保护的成员，但不能直接访问基类中的私有成员。

》》》》预编译头，看视频吧
（【【72】【Cherno C++】【中字】C++的预编译头文件】
https://www.bilibili.com/video/BV1eu411f736/?share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769）

```
    links { "LibraryA", "LibraryB" }

-- 定义构建规则
filter "system:windows"
  defines { "WINDOWS" }

filter "system:linux"
  defines { "LINUX" }

//7. ".." 是字符串连接操作符。
local str1 = "Hello"
local str2 = "World"
local result = str1 .. " " .. str2  -- 将两个字符串连接在一起，中间加
print(result)  -- 输出：Hello World
```

》》》》premake脚本中的 include

作用：： 通常情况下，在 Premake 脚本中使用 include 语句的主要目的是为了包含并执行指定目录下的 Premake 脚本文件。

解释：：

当你在外部的一个 Premake 脚本中写下 include "Nut/vendor/GLFW" 时，这一句实际上是告诉 Premake 构建系统去包含并执行 Nut/vendor/GLFW 目录下的 premake5.lua 文件。这里的含义是引入这个目录下的 Premake 脚本，并将其中定义的项目配置等内容合并到当前的 Premake 脚本中。

虽然 GLFW 目录下可能包含很多文件，但在构建系统中，通常会有一个用于配置和管理该库的专门的构建脚本（比如 premake5.lua）。通过在外部 Premake 脚本中使用 include "Nut/vendor/GLFW"，你实际上是在告诉 Premake 去处理 GLFW 这个目录作为一个整体，而不需要手动列出其中的所有文件。

实际逻辑：：在 Premake 中的 include 语句并不是简单的文本替换或复制粘贴操作。当执行 include 语句时，Premake 会实际上去加载并执行指定目录下的 Premake 文件，并将其中定义的项目配置、编译选项等内容合并到当前的 Premake 上下文中。

》》》》占位符格式化输出信息

eg.

NUT_CORE_INFO("Creating window: {0} ({1} * {2})", props.Title, props.Width, props.Height); 就是在使用占位符来格式化输出信息。

{0}，{1}，{2}分别代表后面的三个参数信息

》》》》》》》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）

》》》》函数指针、std::bind 返回的对象、std::function 返回的对象和 lambda 表达式都可以用来表示函数，但它们之间有一些区别

1.函数指针（Function Pointers）：

概念：	函数指针是指向函数的指针，可直接调用目标函数。 使用函数指针可以直接传递函数地址，或者调用函数。
适用情况：	适用于简单的函数调用，无需状态或额外参数。

2.std::bind：

概念：	std::bind 可以用来将成员函数、自由函数、函数对象等绑定到一个函数对象上，并延迟执行。 可以绑定函数、成员函数、lambda 表达式等，以及提供额外参数。
适用情况：	通常用于创建包装了函数调用和参数的可调用对象。适用于需要延迟执行函数调用或传递额外参数的情况。

3.std::function：

概念：	std::function 是一个通用的可调用对象包装器，可以容纳任意可调用实体（函数指针、函数对象、成员函数指针、 lambda 等）。可以像函数指针一样调用 std::function 对象。
适用情况：	通常用于需要在运行时确定要调用的函数的情况，或者需要存储不同类型的可调用对象。

4.Lambda 表达式：

概念：	Lambda 表达式是一种用于定义匿名函数的语法，可以捕获外部变量。 可以直接在需要函数的地方定义并使用，比较灵活。
适用情况：	适用于需要编写临时的、较短小的函数，以及需要捕获外部作用域变量的情况。

使用情况：

1.使用函数指针	当您只需要简单地传递函数地址或进行函数调用时。
2.使用 std::bind	当您需要绑定函数及其参数，并且可能需要延迟执行函数。
3.使用 std::function	当您需要一个通用的可调用对象容器，并且希望能够存储不同类型的可调用实体。
4.使用 lambda 表达式	当您需要编写临时的、较短小的函数，以及需要捕获外部作用域变量的情况。

》》》》封装的演示：

```
#include <iostream>
#include <stdexcept>

// 原始函数，模拟需要进行权限验证的功能
void sensitiveOperation( auto args )
{
    std::cout << "Sensitive operation performed" << std
    //Or some codes for operate
}

// 用户类，用于存储用户权限信息
class User {
public:
    bool hasPermission(const std::string& permission) {
        // 检查用户是否具有特定权限
        // 这里简化为直接返回 true
        return true;
    }
};

// 权限验证包装器
template <typename Func>
auto permissionWrapper(Func func, const User& user)
{
    return [ func,  &user ]( auto args ) {
```

》》》》封装和包装

1.概念：封装（Encapsulation）	指的是将数据和行为（方法）捆绑在一个单元中，并对外部隐藏对象的内部状态（实现细节），只通过公共接口来访问和操作对象。
目的：	封装提供了一种保护对象状态的机制，可以有效地控制对象的访问权限。更容易维护。
2.概念：包装（Wrapper）	用于将一个对象或函数封装在另一个对象或函数中，以提供额外的功能或修改原始对象或函数的行为，同时保持原始接口不变。
目的：	包装器通常用于添加额外的功能、修改行为或者适应不同的需求，而不会改变原始对象或函数的核心逻辑。

区别：

封装更侧重于“隐藏对象的内部细节和提供公共接口”，以实现数据和行为的封装；

包装则更侧重于在不改变接口的情况下，“为对象或函数添加额外的功能或修改其行为”。

演示： ---->

》》》》关于emplace 和 push

emplace & push

emplace_back & push_back

在用法和语义上都是一样的，之不过push会额外复制一次用来传输

》》》》关于emplace 和 push

emplace & push

emplace_back & push_back

在用法和语义上都是一样的，之不过push会额外复制一次用来传输

》》》》emplace 和 emplace_back

emplack_back( )	只接受一个参数，就是传入的元素，因为会直接插入到vector的末尾
emplace( )	接受两个参数，一个是插入的位置，第二个是传入的元素。

但是emplace有两个要点：

1.插入的元素的位置在填入的位置之前。

2.插入之后，会返回一个值，该值为新插入的元素的位置。

》》》》vector.end( ) 是什么? std::find( ) 的返回值在什么时候与其相同?

1.m_Layers.end( ) 返回的是指向容器中最后一个元素之后的位置（即尾后迭代器），而不是指向空的指针。

2.当std::find( ) 没有找到所查证的元素时。就会返回vector.end( ) 相同的值。所以

```
auto it = std::find(m_Layers.begin(), m_Layers.end(), layer);
    if (it != m_Layers.end()) {
        XXXXXX
    }
```

多可以用来在vector中从头到尾的查询一个值，并做出判断。

》》》》函数声明的最后const是什么意思?

在这个成员函数不允许修改成员变量的值

》》》》序列化是啥?

序列化是指将数据结构或对象转换为一种特定格式，以便于存储或传输，同时也可以在需要时将其恢复为原始的数据结构或对象的过程。

通常涉及以下几个方面：

将数据结构转换为字节流：将数据结构中的各个字段按照一定规则编码成字节流，以便于存储或传输。

包含足够的信息：序列化的数据应该包含足够的信息，以便在反序列化时能够准确地重新构建原始的数据结构或对象。

》》》》.hpp和.h文件的区别

目的：

为了更清晰地表明文件是用于 C++ 的代码还是传统的 C 代码。

.h 文件通常用于传统的 C/C++ 头文件

而 .hpp 文件则有时被用于 C++ 头文件，特别是在涉及到 C++ 的特性或标准库时。

》》》》什么是 constexpr ?

概念：

constexpr是C++11引入的一个关键字，用于声明可以在编译时求值的常量表达式。它可以用于变量、函数、构造函数等的声明中。

变量：

通过在变量声明前加上constexpr关键字，可以将其定义为常量表达式。

必须满足以下条件：

初始化表达式在编译时就能计算出结果；

变量类型必须是字面类型（literal type），即不能包含用户自定义类型的成员函数，且所有非静态成员必须也是字面类型。

```
eg.
    constexpr int k = 10;
    constexpr double pi = 3.1415926;
```

函数：

通过在函数声明前加上constexpr关键字，可以将其定义为常量表达式。

函数必须满足以下条件：

函数体中只能包含一行返回语句；

返回值必须是字面类型；

所有参数必须是字面类型。

```
eg.
    constexpr int factorial(int n) {
        return (n == 0) ? 1 : n * factorial(n - 1);
    }
    constexpr int result = factorial(5); // 编译时就能计算出结果

```

构造函数：

通过在构造函数声明前加上constexpr关键字，可以将其定义为常量表达式构造函数。

类必须满足以下条件：

类中的所有成员变量和成员函数都是字面类型；

构造函数只能有一个参数列表；

构造函数中只能使用一行赋值语句对成员变量进行初始化。

```
// 权限验证包装器
template <typename Func>
auto permissionWrapper(Func func, const User& user)
{
    return [ func,  &user ]( auto args ) {
        if (user.hasPermission("execute")) {
            // 具有权限，调用原始函数
            func( args );
        } else {
            // 没有权限，抛出异常或返回错误信息
            throw std::runtime_error("Permission denied
        }
    };
}

int main()
{
    // 创建一个用户
    User currentUser;

    // 使用权限验证包装器
    auto wrappedFunc = permissionWrapper(sensitiveOpera

    // 调用被包装的函数
    try {  wrappedFunc( //args or none );  }
    catch (const std::exception& e) {  std::cerr << "Er

    return 0;
}

```

```
eg.
    class Vector {
    public:
        constexpr Vector(double x, double y) : x_(x), y_(y) {}
        constexpr double x() const { return x_; }
        constexpr double y() const { return y_; }
    private:
       double x_;
       double y_;
    };
    constexpr Vector v(1.0, 2.0); // 编译时就能创建对象
```

作用：

在使用constexpr时，最重要的是确保定义的常量表达式可以在编译时求值。

》》》》

int func();

int func() const;

const int& func();

const int& func() const; 的区别。

- **1.int func();**

非常量成员函数.

1.可以修改对象的成员变量

2.返回成员变量的拷贝

- **2.int func() const;**

成员函数

1.不会修改对象的成员变量

2.但是它返回的是成员变量的拷贝，而不是引用.(对返回值的修改不会影响到原始对象的成员变量。

- **3.const int& func();**

可以是一个全局函数，静态函数，或者是一个类的成员函数

1.在对应情况下，表示可以修改静态变量、全局变量或成员变量

2.返回一个对静态变量或者全局变量的常量引用

- **4.const int& func() const;**

成员函数

1.不会修改对象的成员变量

2.返回一个对成员变量的常量引用

》》》》**包含库目录和链接** 这两个词的概念与关系：

包含目录

作用：	指定编译器在编译时搜索头文件（.h 或 .hpp 文件）的路径。
机理：	编译器在编译源文件时使用这些目录来找到包含的头文件，例如 #include "foo.h"。头文件通常包含函数声明、类定义和宏定义等，不包括具体的实现。
效果：	允许源文件访问库的头文件，编译时能找到并正确使用库的接口（例如函数声明和类定义）。

链接

作用：	将编译生成的目标文件（.o 或 .obj 文件）与库文件（.lib、.a 或 .so、.dll 文件）合并，生成最终的可执行文件或库文件。
机理：	链接器将目标文件和库文件中的符号（例如函数和变量）解析并连接起来，解决未定义的符号引用。库文件可以是静态库（编译时链接）或动态库（运行时链接）。
效果：	在编译生成的目标文件和库文件中，解析和连接实际的实现代码，生成可以执行的程序或库。

典型使用：

1. 包含目录：在编译阶段指定，使编译器能够找到库的头文件。
2. 链接：在链接阶段指定，使链接器能够将目标文件与库文件正确结合，生成最终的可执行文件。

》》》》C语言占位符详见：

C语言风格常见格式化占位符：

%s：用来表示一个字符串。它要求后续参数是一个 char* 类型的字符串。	例如：
	const char* name = "Entity1";
	printf("Hovered Entity: %s\n", name);
	// 输出：Hovered Entity: Entity1

%d：用来表示一个整数（int 类型）。它用于插入一个整数值。	例如：
	int id = 42;
	printf("Hovered Entity ID: %d\n", id);
	// 输出：Hovered Entity ID: 42

%f：用来表示一个浮点数。	例如：
	float x = 10.5f;
	printf("Position: %f\n", x);
	// 输出：Position: 10.500000

%x：用来表示一个十六进制数。	例如：
	int num = 255;
	printf("Hexadecimal: %x\n", num);
	// 输出：Hexadecimal: ff

%p：用来表示一个指针（地址）。	例如：
	int* ptr = &id;
	printf("Pointer address: %p\n", ptr);
	// 输出类似：Pointer address: 0x7fffd60fdbec

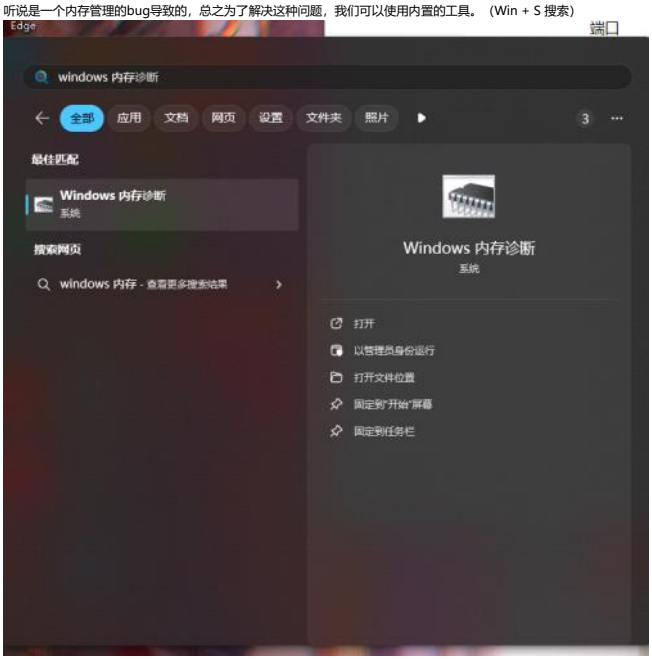
%c：用来表示一个字符。	例如：
	char ch = 'A';
	printf("Character: %c\n", ch);
	// 输出：Character: A

%u：用来表示一个无符号整数。	例如：
	unsigned int val = 123;
	printf("Unsigned value: %u\n", val);
	// 输出：Unsigned value: 123

Win11的内存管理有Bug，有时候刚刚开机，内存就已经占用了百分之五十甚至以上，故使用内置工具修复问题。

而且下载了一个管理并清除内存的轻量化软件: Mem Reduct,可以在任务栏监视内存使用情况，也可以设置每隔一段时间自动清除内存。非常方便。
下载链接:
（henrypp/memreduct: Lightweight real-time memory management application to monitor and clean system memory on your computer. (github.com)）

问题描述
》》》》关于 Windows 11 不堪的内存状况
用久了Win11总是出现内存占用高的状况，可明明没有打开高消耗的应用。
听说是一个内存管理的bug导致的，总之为了解决这种问题，我们可以使用内置的工具。（Win + S 搜索）

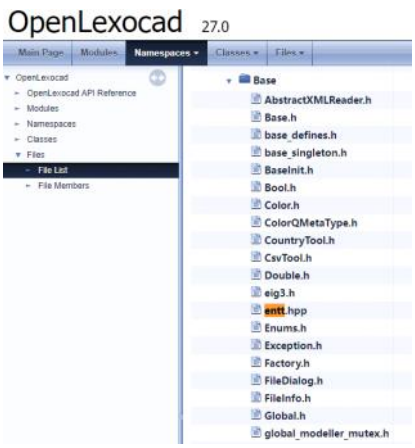


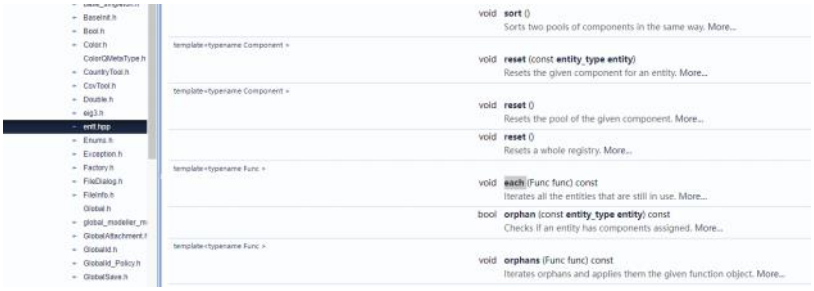
这可能会耗费十分钟的时间，电脑会蓝屏自诊。

点击立即重启即可
这个过程需要很长的时间，我的大概10分钟左右，需要总体测试的任务进度到100%即可。

开机之后发现，内存立马降低，我的是16GB的内存，之前开机立马到70%，现在开机20%，恢复正常。

》》》》doxygen 下载doxygen: https://www.doxygen.nl/
不知道怎么的我翻阅到entt先前的文档，然后查阅了先前的设计。找到了这个阴魂不散的each





不过这好像是entt.hpp之前版本的数据，所以each()的使用方法参考性不大。于是我决定下载一个doxygen看看能不能识别一下现如今最新的entt.hpp.

下载网址： https://www.doxygen.nl/
下载以及使用参考：(感谢愿意分享经验的从业者们，也感谢 @skypjack 的 enTT 代码.)

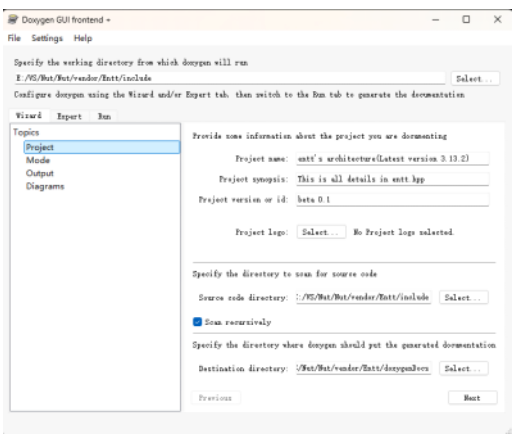
@请叫我Axin:（doxygen配置与使用的视频演示）
Bilibili（https://www.bilibili.com/video/BV1ZE411F7kW/?p=4&share_source=copy_web&vd_source=ca2feff7d155a2579964dfa2c3173769)

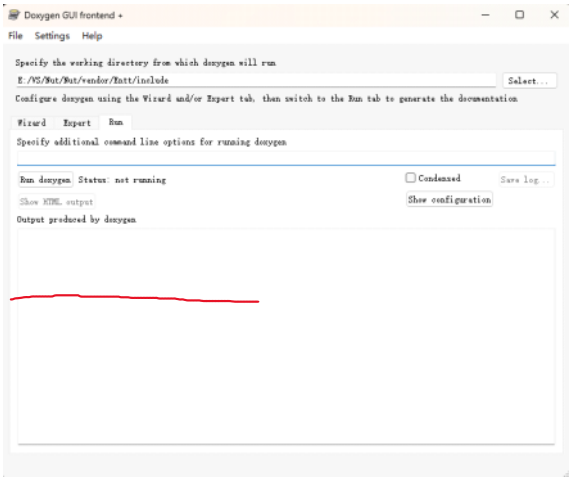
@doxygen官方文档： https://www.doxygen.nl/manual/starting.html

@Dariusz Zyzański （文档演示以及简单的教学）：Doxygen C++ documentation for complete beginners

@Christoph Schlosser（VSCode中使用 doxygen 的集成插件）： https://marketplace.visualstudio.com/items?itemName=cschlosser.doxdocgen

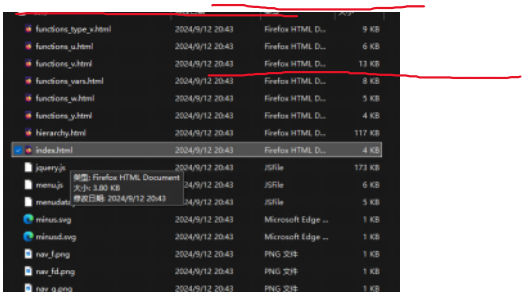
一路默认之后，在开始菜单处打开GUI。



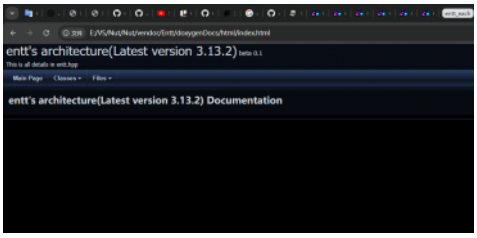


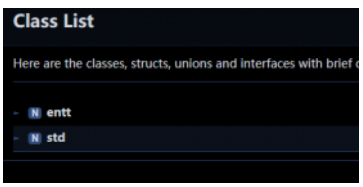
Specify the working directory from which Doxygen will run:	**作用**：指定 Doxygen 执行时的工作目录。这个目录是 Doxygen 扫描源文件和配置文件的起点。 **填写方式**：填写你希望 Doxygen 从中读取源代码和配置文件的目录路径。例如，如果你的源代码和 Doxyfile 在 project/src 目录下，你应指定为 project/src。
Specify the directory where Doxygen should put the generated documentation:	**作用**：指定 Doxygen 生成的文档应该保存到的输出目录。文档生成后将会被存储在这个目录中。 **填写方式**：填写你希望保存生成文档的目录路径。比如，你可以填写 project/docs，Doxygen 会在此目录中创建文档文件。
Specify additional command line options for running Doxygen:	**作用**：允许你为 Doxygen 执行提供额外的命令行选项。这些选项可以用来调整 Doxygen 的行为或功能。 **填写方式**：填写你想要传递给 Doxygen 的额外参数。例如，你可以指定 -d 以开启调试模式，或者 -s 以生成静态文档。你可以根据需要添加其他 Doxygen 命令行选项。

随后敲击 **Run doxygen，将会在你指定的路径生成可打开的 HTML 文件。**使用浏览器打开即可。

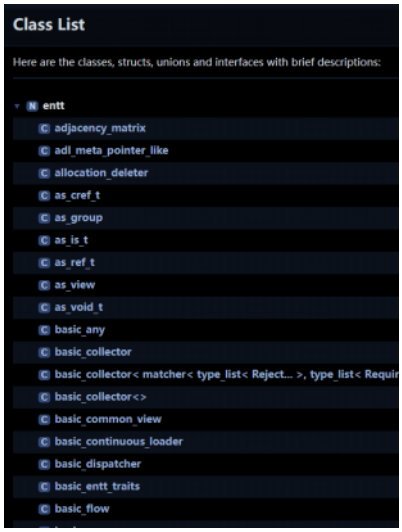


进入浏览器，按照指示选项卡打开，查看

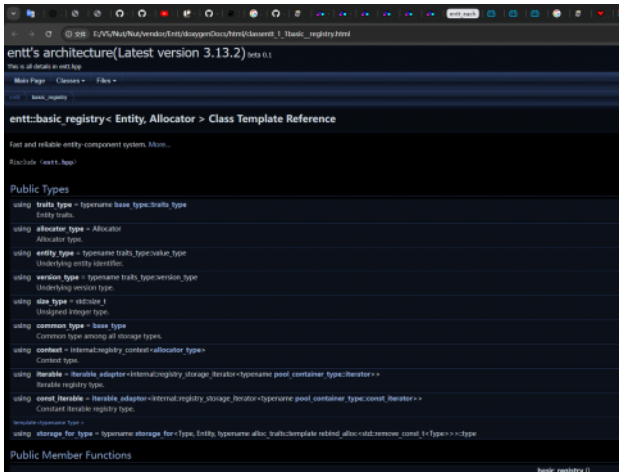




在所有class中，我们找到basic_registry,并打开。



在这里，你可以查找类型、成员函数等等



可以看到，最新版本的 entt 中，entt::basic_registry 并没有 each() 函数