## 49 Using Libraries in C++ (Static Linking)

C++中使用库 (静态链接)

如果你用过其它语言,如python,C#等,添加库是一项非常简单的任务。你可能用的是包管理器等等。但是在C++这里好像任何人都有问题,看上去很难但实际很简单。

本篇将以二进制文件的形式进行链接,而不是获取实际依赖库的源代码并自己进行编译,确切来说,就是GLFW库

在你的实际项目或你想链接的库中,二进制文件可能会不可用,所以实际上你可能会被迫自己去构建它,尤其是对于Mac和 Linux用户。在一个更加专业的大项目中,有足够时间的话,还是自己去编译好,因为这有助于调试,如果你想修改库就稍微改一下,这种情况下Cherno就比较喜欢在VS项目中用源代码自己编译。

## 1.32位还是64位?

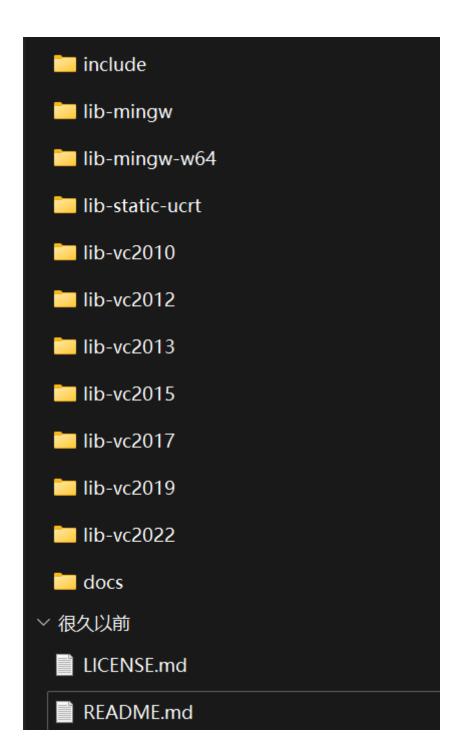
#### Download GLFW

我想要32位的二进制文件还是64位的?

这与你实际的操作系统没有任何关系,如果你用的是win10,64位比特的操作系统,并不意味着你应该获取64位的二进制文件,你应该根据你的目标应用程序来选择。

所以如果我编译我的应用程序,作为X86也就是win32程序,那么我就要32位的二进制文件;如果我在编译一个64位的应用程序,我就要64位的二进制文件——一定要把它们匹配起来,因为如果不匹配的话它们无法进行链接。

这里选择32位的下载,解压后可以看到这是一种典型的文件布局(C++库的典型组织结构)。库通常包含两部分,includes和 library,包括目录和库目录。



include (包含目录) 是一堆我们需要的头文件,这样我们就可以实际使用预构建的二进制文件中的函数,然后lib目录有那些预先构建的二进制文件。

这里通常有两部分,有动态库和静态库(但并不是所有库都为你提供了这两种库,但GLFW库为你提供了两种),你可以选择是 静态链接还是动态链接。

# 2. 静态 v.s. 动态链接

这里先简单讲下区别:

## 静态链接

静态链接意味着这个库会被放到你的可执行文件中(它在你的exe文件内部,或者其它操作系统下的可执行文件内)。

#### 动态链接

动态链接库是在运行时被链接的额,所以你仍有一些链接,你可以选择在程序运行时装载动态链接库,有一个叫loadLibrary的函数,你可以在WindowsAPI中使用它作为例子。它会载入你的动态库,可以从中拉出函数然后开始调用。你也可以在应用程序启动时加载你的dll文件,这就是你的Dynamic Link Library(动态链接库)。

所以主要的区别就是:库文件是否被编译到exe文件中,或链接到exe文件中,还是一个在运行时单独的文件,你需要把它放在你的exe文件旁边或某个地方,然后你的exe文件可以加载它。因为这种依赖性,你需要把exe文件和dll文件弄在一起。

所以通常喜欢用静态的。静态链接在技术上更快,因为编译器或链接器实际上可以执行链接时优化之类的。静态链接在技术上可以产生更快的应用程序,因为有几种优化方法可以应用,因为我们知道在链接时要链接的函数。而对于动态库,我们不知道会发生什么而必须保持它的完整,当动态链接库被运行时的程序装载时,程序的部分将被补充完整。

所以通常情况下,静态链接是更好的选择。



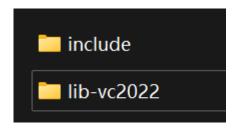
## 3. 静态链接实例

我们在Solution文件所在的目录新建一个Dependencies文件夹来容纳依赖项,里面创建一个GLFW文件夹,在这里面放进刚才解压的文件。



这里面附加了很多使用不同编译器编译出来的库文件,比如mingw等,以及各个版本的VS,这里就选择最新的一个。选择哪个其实并不重要,因为说到底它们只是一种编译过的二进制文件,其中任何一个都可以工作。但我们想用最兼容我们当前tool chain(工作链)的。

这里把include和选择的版本放进来。

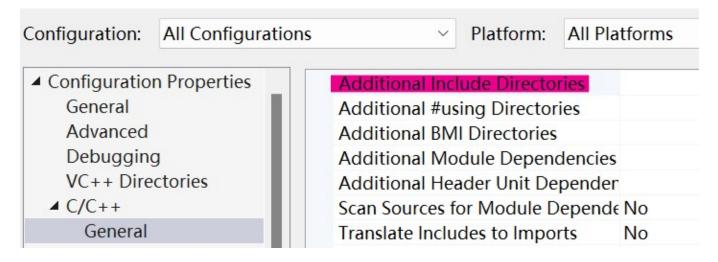


打开lib-vc2022,可以看到如下文件:



- glfw3.dll是一种运行时动态链接库,我们在运行时动态链接时会用到它。
- **glfw3dll.lib** 实际上是一种与glfw3.dll是一起用的静态库,这样我们就不用实际询问dll我们有没有一堆指向所有这些函数的函数指针。这就是说这个lib文件实际包含了glfw3.dll中所有函数、符号的位置,所以我们可以在编译时链接它们。如果我们没有这个lib文件,我们依然可以使用dll文件,不过我们需要通过**函数名**来访问dll文件内的函数(比如GLFW\_INIT),但这个lib文件包含了所有这些函数的位置,链接器可以直接链接到它们。
- **glfw3.lib** 可以发现这个静态链接库比其它的大很多,如果我们不想要编译时链接,我们就链接这个lib,如果这样的话, 我们在exe运行时就不需要这个dll文件。

首先,我们在C/C++下的General,Additional Include Directories(附加的包含目录)中指定附加的包含目录。别忘了注意要指定你的配置和平台,这里就设置为All Configuration。



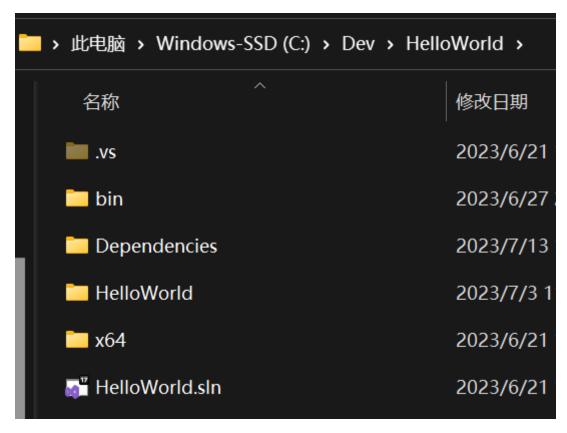
包含目录就是这个include文件夹的实际目录:



我可以直接复制这个目录,但很明显可以看到它是类似这样的格式:

C:\Dev\HelloWorld\Dependencies\GLFW\include

这个是特指自己电脑当前的路径,如果有人从GitHub或者别的地方克隆了这个,如果还是这个路径的话代码就不会通过编译。 我们真正想要的是一个相对于这个实际目录的路径,而HelloWorld是我的解决方案的目录:



从这个解决方案开始,可以用VS的一个macro (宏指令)

Additional Include Directories	\$(SolutionDir)
Additional #using Directories	Additional Include Director
Additional BMI Directories	
Additional Module Dependencie	
Additional Header Unit Depende	
Scan Sources for Module Depend	\$(SolutionDir)
Translate Includes to Imports	
Debug Information Format	
Support Just My Code Debuggin	
Common Language RunTime Sur	Evaluated value:
Consume Windows Runtime Exte	C:\Dev\HelloWorld\
Suppress Startup Banner	

\$(SolutionDir)Dependencies\GLFW\include //注意宏指令自带一个'\'

我们现在能做的就是包含这个目录(include)相对路径下的相关文件,这个目录包含了GLFW文件夹,里面有个文件叫glfw3.h,所以我们可以输入:

因为这是一个编译器指定的包含路径,我也可以用尖括号#include <GLFW/glfw3.h>。

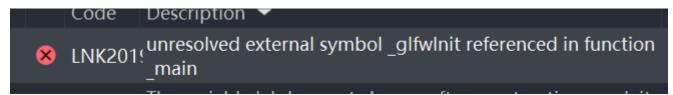
这里就引出了用引号还是尖括号的问题,实际上没有区别,因为如果是引号的话会先检查相对路径,如果它没找到任何相对于这个文件的东西,也就是相对于main.cpp,它就会去检查编译器的include路径。

#### 可参考的选择方法:

- 如果这个源文件在visual studio中(在解决方案中的某个地方,也许在另一个项目中,并不在乎,只要在这个Solution里面),那就用 " ".
- 如果它是一个完全的外部依赖或者外部的库,不在visual studio中和我的实际解决方案一起编译,那就用 < > 来表示它实际上是外部的。

int a = glfwInit(); //编译成功, 但build失败

显示unresolved external symbol (未解析的外部符号)



这意味着你并没有链接到实际的库,链接器找不到这个glfwlnit函数。

修改Linker设置,将lib文件的目录设置好,来指定一个库目录。

	meremental Enik Database i ne	#(IIICDII)#(Tai geti tai iie).iik
▷ C/C++	Suppress Startup Banner	Yes (/NOLOGO)
▲ Linker	Ignore Import Library	No
General	Register Output	No
Input	Per-user Redirection	No
Manifest File	Additional Library Directories	\$(SolutionDir)Dependencies\GLFW\lib-vc2022
Debugging	Link Library Dependencies	Yes
Debugging System	Link Library Dependencies Use Library Dependency Inputs	Yes No
33 3		

#### 再指定相对于那个库目录的库文件名称

▲ Configuration Properties
General
Advanced
Debugging
VC++ Directories
▷ C/C++
▲ Linker
General
Input

Additional Dependencies	glfw3.lib;\$(CoreLi
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

我们把结果打印给控制台:

```
#include <iostream>
#include <GLFW/glfw3.h>

int main()
{
    int a = glfwInit();
    std::cout << a << std::endl; // 1 ,说明运行成功了

    std::cin.get();
}
```

如果去掉 include, 输入glfwlnit的实际声明:

```
#include <iostream>
//#include <GLFW/glfw3.h>

int glfwInit();

int main()
{
   int a = glfwInit();
   std::cout << a << std::endl;
   std::cin.get();
}</pre>
```

这个想法是可行的,但是在这个特定的例子中我们得到了一个错误。因为GLFW库实际上是一个C语言库,我们在这里用的是C++,混淆了名字(Name mangling)。

我们需要在函数声明前面加上 extern "C":

```
c++
extern "C" int glfwInit();
```

意思是保持这个名字的原貌,因为你会链接到一个用C语言建立的库。运行成功也输出了"1",和我们在有头文件的时候得到的结果一样。

所以记住,库的链接也好,头文件也好,并没有什么不得了的魔法,它们只是将系统的所有组成部分连接在一起。头文件通过提供声明,告诉我们哪些函数是可用的,库文件为我们提供了定义,这样我们就可以链接到那些函数,并在C++中调用函数时执行正确,