Dynamic-Link Library (DLL)

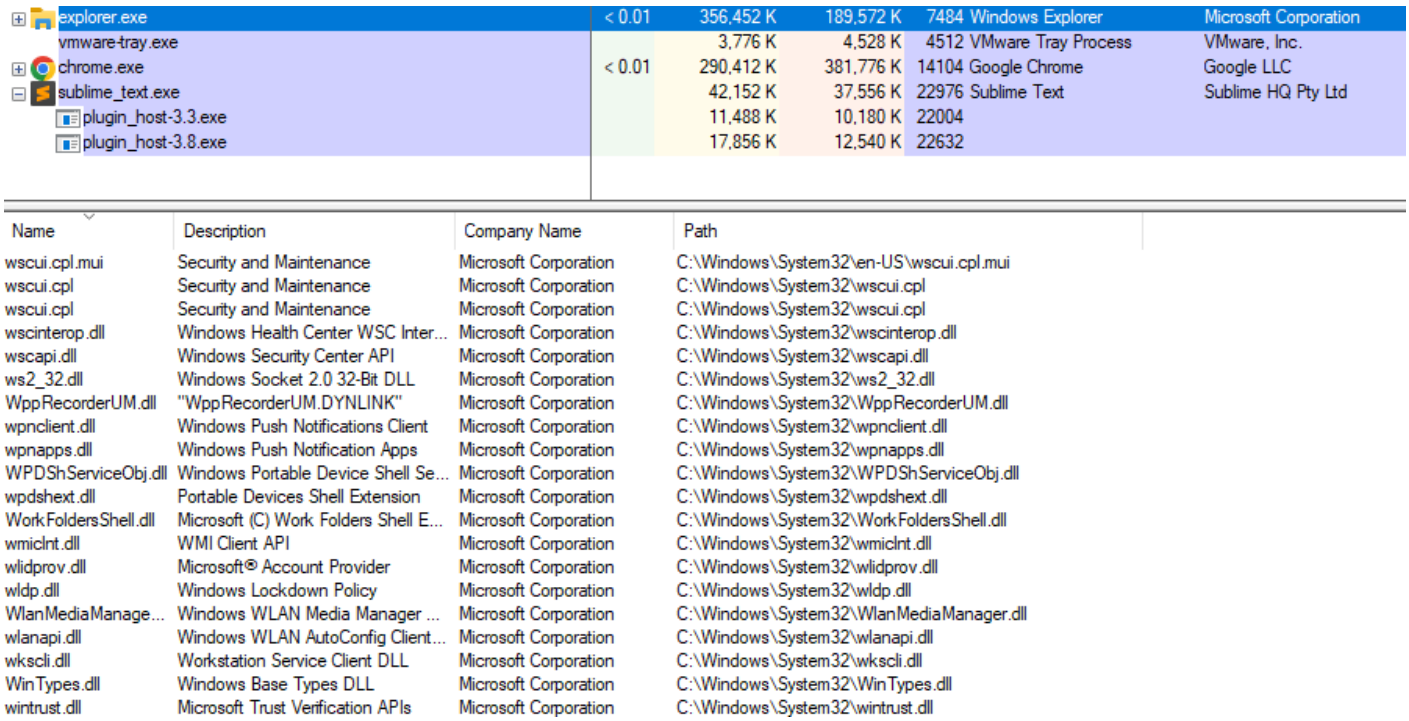
Introduction

Both `.exe` and `.dll` file types are considered portable executable formats but there are differences between the two. For example, one major difference that can be immediately noticed is that `.exe` files can be executed by being double clicked whereas the same cannot be done onto `.dll` files. This module will outline additional differences between the two file types.

What is a DLL?

DLLs are shared libraries of executable functions or data that can be used by multiple applications simultaneously. They are used to export functions to be used by a process. Unlike EXE files, DLL files cannot execute code on their own. Instead, DLL libraries need to be invoked by other programs to execute the code. As previously mentioned, the `CreateFileW` is exported from `kernel32.dll`, therefore if a process wants to call that function it would first need to load `kernel32.dll` into its address space.

Some DLLs are automatically loaded into every process by default since these DLLs export functions that are necessary for the process to execute properly. A few examples of these DLLs are `ntdll.dll`, `kernel32.dll` and `kernelbase.dll`. The image below shows several DLLs that are currently loaded by the `explorer.exe` process.

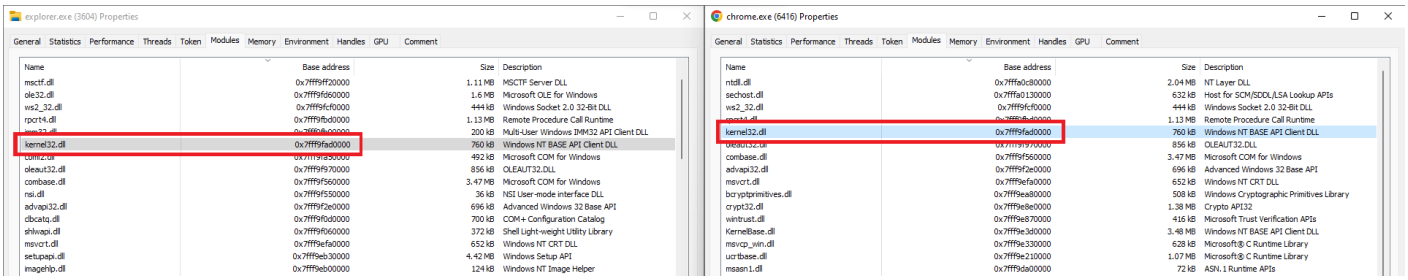


explorer.exe	< 0.01	356,452 K	189,572 K	7484 Windows Explorer	Microsoft Corporation
vmware-tray.exe		3,776 K	4,528 K	4512 VMware Tray Process	VMware, Inc.
chrome.exe	< 0.01	290,412 K	381,776 K	14104 Google Chrome	Google LLC
sublime_text.exe		42,152 K	37,556 K	22976 Sublime Text	Sublime HQ Pty Ltd
plugin_host-3.3.exe		11,488 K	10,180 K	22004	
plugin_host-3.8.exe		17,856 K	12,540 K	22632	

Name	Description	Company Name	Path
wscui.cpl.mui	Security and Maintenance	Microsoft Corporation	C:\Windows\System32\en-US\wscui.cpl.mui
wscui.cpl	Security and Maintenance	Microsoft Corporation	C:\Windows\System32\wscui.cpl
wscui.cpl	Security and Maintenance	Microsoft Corporation	C:\Windows\System32\wscui.cpl
wscinterop.dll	Windows Health Center WSC Inter...	Microsoft Corporation	C:\Windows\System32\wscinterop.dll
wscapi.dll	Windows Security Center API	Microsoft Corporation	C:\Windows\System32\wscapi.dll
ws2_32.dll	Windows Socket 2.0 32-Bit DLL	Microsoft Corporation	C:\Windows\System32\ws2_32.dll
WppRecorderUM.dll	"WppRecorderUM.DYNLINK"	Microsoft Corporation	C:\Windows\System32\WppRecorderUM.dll
wpnclient.dll	Windows Push Notifications Client	Microsoft Corporation	C:\Windows\System32\wpnclient.dll
wpnapps.dll	Windows Push Notification Apps	Microsoft Corporation	C:\Windows\System32\wpnapps.dll
WPDShServiceObj.dll	Windows Portable Device Shell Se...	Microsoft Corporation	C:\Windows\System32\WPDShServiceObj.dll
wpdshext.dll	Portable Devices Shell Extension	Microsoft Corporation	C:\Windows\System32\wpdshext.dll
WorkFoldersShell.dll	Microsoft (C) Work Folders Shell E...	Microsoft Corporation	C:\Windows\System32\WorkFoldersShell.dll
wmiclnt.dll	WMI Client API	Microsoft Corporation	C:\Windows\System32\wmiclnt.dll
wlidprov.dll	Microsoft® Account Provider	Microsoft Corporation	C:\Windows\System32\wlidprov.dll
wldp.dll	Windows Lockdown Policy	Microsoft Corporation	C:\Windows\System32\wldp.dll
WlanMediaManage...	Windows WLAN Media Manager ...	Microsoft Corporation	C:\Windows\System32\WlanMediaManager.dll
wlanapi.dll	Windows WLAN AutoConfig Client...	Microsoft Corporation	C:\Windows\System32\wlanapi.dll
wkscli.dll	Workstation Service Client DLL	Microsoft Corporation	C:\Windows\System32\wkscli.dll
WinTypes.dll	Windows Base Types DLL	Microsoft Corporation	C:\Windows\System32\WinTypes.dll
wintrust.dll	Microsoft Trust Verification APIs	Microsoft Corporation	C:\Windows\System32\wintrust.dll

System-Wide DLL Base Address

The Windows OS uses a system-wide DLL base address to load some DLLs at the same base address in the virtual address space of all processes on a given machine to optimize memory usage and improve system performance. The following image shows `kernel32.dll` being loaded at the same address (`0x7fff9fad0000`) among multiple running processes.



Why Use DLLs?

There are several reasons why DLLs are very often used in Windows:

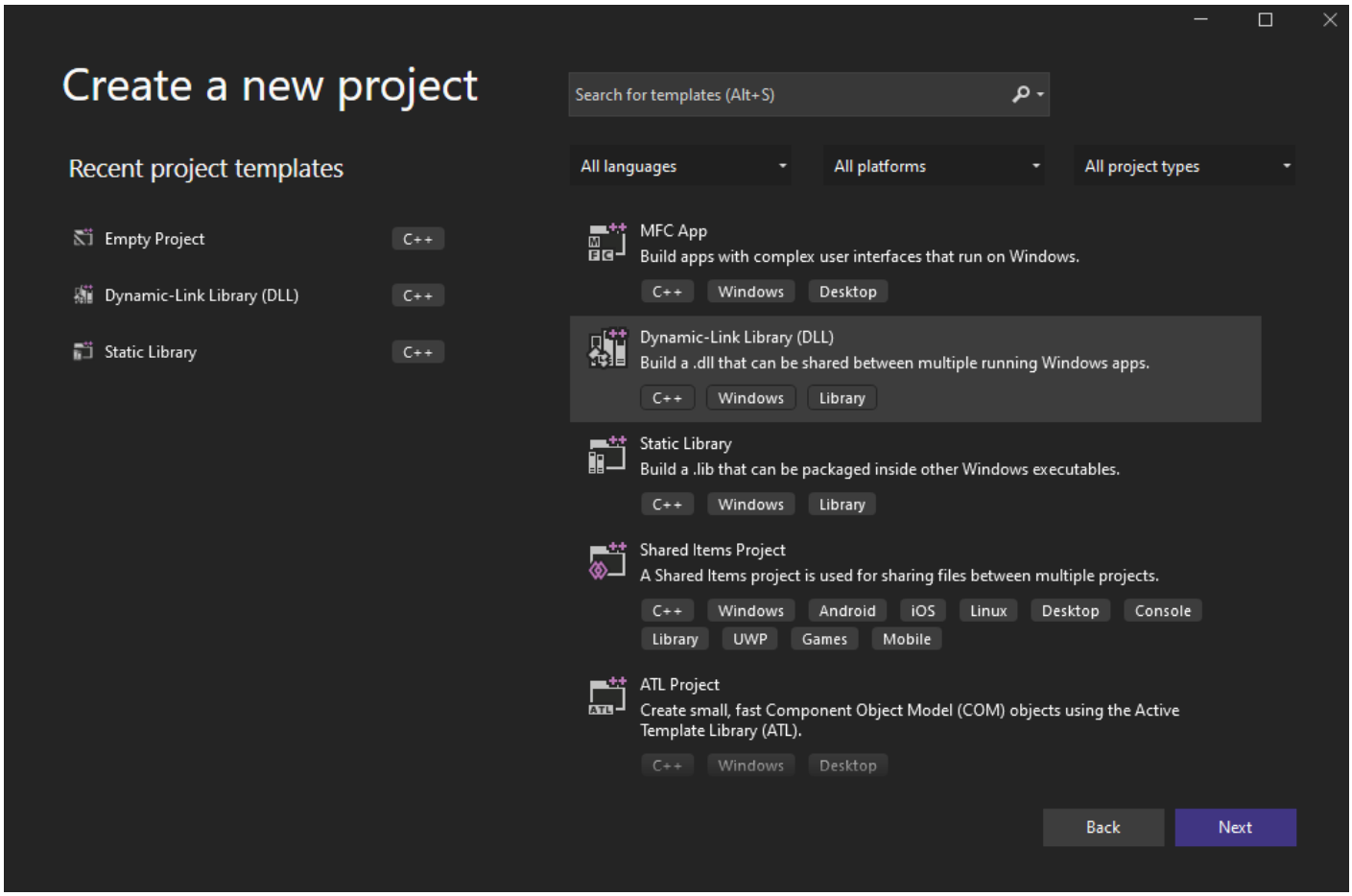
1. **Modularization of Code** - Instead of having one massive executable that contains the entire functionality, the code is divided into several independent libraries with each library being focused on specific functionality. Modularization makes it easier for developers during development and debugging.

2. **Code Reuse** - DLLs promote code reuse since a library can be invoked by multiple processes.

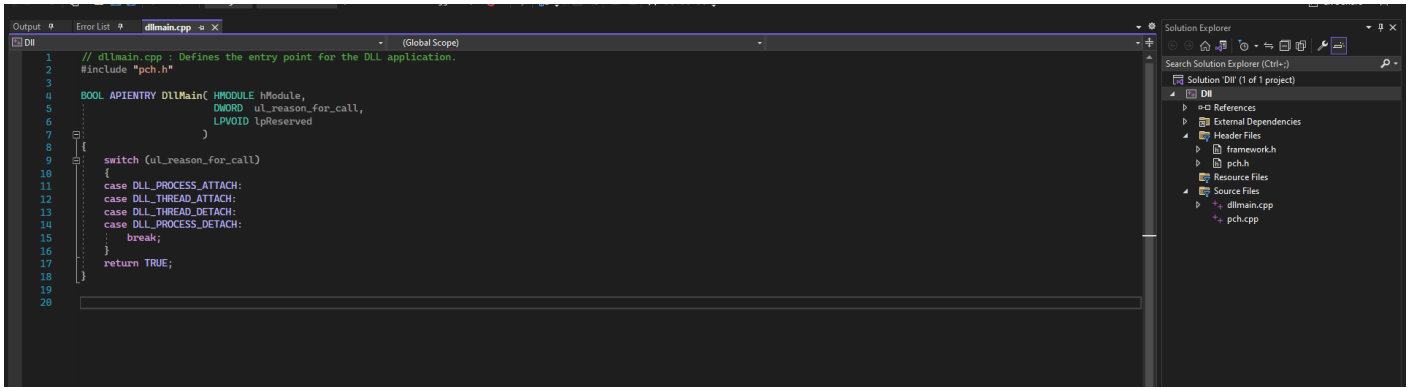
3. **Efficient Memory Usage** - When several processes need the same DLL, they can save memory by sharing that DLL instead of loading it into the process's memory.

Creating a DLL File With Visual Studio

To create a DLL file, launch Visual studio and create a new project. When given the project templates, select the `Dynamic-Link Library (DLL)` option.



Next, select the location where to save the project files. When the project has been saved, `dllmain.cpp` should appear with the default DLL code.



DLL Entry Point

Recall that DLLs are loaded by applications (e.g. `.exe` files). Therefore, DLLs can specify an entry point function that executes code when a certain action occurs. There are 4 possibilities for the entry point being called:

- `DLL_PROCESS_ATTACH` - A process is loading the DLL.
- `DLL_THREAD_ATTACH` - A process is creating a new thread.

- `DLL_THREAD_DETACH` - A thread exits normally.
- `DLL_PROCESS_DETACH` - A process unloads the DLL.

Exporting a Function

DLLs can export functions that can then be used by the calling application. To export a function it needs to be defined using the keywords `extern` and `__declspec(dllexport)`. An example exported function `HelloWorld` is shown in the `sampleDLL.dll` file below.

```
////// sampleDLL.dll //////

#include <Windows.h>

// Exported function
extern __declspec(dllexport) void HelloWorld(){
    MessageBoxA(NULL, "Hello, World!", "DLL Message", MB_ICONINFORMATION);
}

// Entry point for the DLL
BOOL APIENTRY DllMain(HMODULE hModule, DWORD  ul_reason_for_call, LPVOID
lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

`HelloWorld` can now be invoked by an external application after loading `sampleDLL.dll` into memory.

Dynamic Linking

It's possible to use the `LoadLibrary`, `GetModuleHandle` and `GetProcAddress` WinAPIs to import a function from a DLL. This is referred to as dynamic linking. This is a method of loading and linking code (DLLs) at runtime rather than linking them at compile time using the linker and import address table. There are several advantages of using dynamic linking; these are documented by Microsoft here.

The upcoming sections walk through the steps of loading a DLL, retrieving the DLL's handle, retrieving the exported function's address, and then invoking the function from an external binary.

Step 1 - Loading a DLL

We're going to switch to an EXE file in this step and the steps that follow. This is because our EXE file will be the one loading `sampleDLL.dll` and invoking the `HelloWorld` function. Therefore, create a new Win32 Console Application and follow along to invoke `HelloWorld`.

Calling a function such as MessageBoxA in an application will force the Windows OS to load the DLL exporting the `MessageBoxA` function into the calling process's memory address space, which in this case is `user32.dll`. Loading `user32.dll` was done automatically by the OS when the process started and not by the code.

However, for custom DLLs such as our `sampleDLL.dll`, the DLL will not be loaded into memory. Since the application doesn't have `sampleDLL.dll` loaded into memory, it would require the usage of the LoadLibrary WinAPI, as shown below.

```
#include <windows.h>

int main() {
    // Load the DLL
    HMODULE hModule = LoadLibraryA("sampleDLL.dll"); // hModule now contain
sampleDLL.dll's handle

}
```

Step 2 - Retrieving a DLL's Handle

Had `sampleDLL.dll` been already loaded into the application's memory, one can retrieve its handle via the GetModuleHandle WinAPI function without leveraging the `LoadLibrary` function.

```
#include <windows.h>

int main() {
    // Attempt to get the handle of the DLL that's already in memory
    HMODULE hModule = GetModuleHandleA("sampleDLL.dll");

    if (hModule == NULL) {
        // If the DLL is not loaded in memory, use LoadLibrary to load it
        hModule = LoadLibraryA("sampleDLL.dll");
    }
}
```

Step 3 - Retrieving a Function's Address

Once the DLL is loaded into memory and the handle is retrieved, the next step is to retrieve the function's address. This is done using the GetProcAddress WinAPI which takes the handle of the DLL that exports the function and the function name.

```
#include <windows.h>

int main() {
    // Attempt to get the handle of the DLL
    HMODULE hModule = GetModuleHandleA("sampleDLL.dll");

    if (hModule == NULL) {
        // If the DLL is not loaded in memory, use LoadLibrary to load it
        hModule = LoadLibraryA("sampleDLL.dll");
    }

    PVOID pHelloWorld = GetProcAddress(hModule, "HelloWorld"); ///
pHelloWorld stores HelloWorld's function address
}
```

Step 4 - Type-casting The Function's Address

Once `HelloWorld`'s address is saved into the `pHelloWorld` variable, the next step is to perform a type-cast on this address to `HelloWorld`'s function pointer. This function pointer is required in order to invoke the function.

```
#include <windows.h>

// Constructing a new data type that represents HelloWorld's function
pointer
typedef void (WINAPI* HelloWorldFunctionPointer)();

int main() {
    // Attempt to get the handle of the DLL
    HMODULE hModule = GetModuleHandleA("sampleDLL.dll");

    if (hModule == NULL) {
        // If the DLL is not loaded in memory, use LoadLibrary to load it
        hModule = LoadLibraryA("sampleDLL.dll");
    }

    PVOID pHelloWorld = GetProcAddress(hModule, "HelloWorld"); ///
pHelloWorld stores HelloWorld's function address

    HelloWorldFunctionPointer HelloWorld =
(HelloWorldFunctionPointer)pHelloWorld;

    return 0;
}
```

Putting It Together - Invoking HelloWorld

This section will now put all the aforementioned steps into one function, called `call()`. The function will essentially perform the following steps:

1. Load `sampleDLL.dll`
2. Retrieve the `HelloWorld` function's address
3. Type-cast `HelloWorld`
4. Invoke `HelloWorld`

Again, this function is being called from our `.exe` program as it is the one loading the DLL and invoking the `HelloWorld` function.

```
#include <windows.h>

// Constructing a new data type that represents HelloWorld's function
pointer
typedef void (WINAPI* HelloWorldFunctionPointer)();

void call() {
    // Attempt to get the handle of the DLL
    HMODULE hModule = GetModuleHandleA("sampleDLL.dll");

    if (hModule == NULL) {
        // If the DLL is not loaded in memory, use LoadLibrary to load it
        hModule = LoadLibraryA("sampleDLL.dll");
    }

        // pHelloWorld stores HelloWorld's function address
    PVOID pHelloWorld = GetProcAddress(hModule, "HelloWorld");


    // Typecasting pHelloWorld to be of type HelloWorldFunctionPointer
    HelloWorldFunctionPointer HelloWorld =
(HelloWorldFunctionPointer)pHelloWorld;

        // Invoke HelloWorld
    HelloWorld();

}
```

Dynamic Linking Example - MessageBoxA

The code below demonstrates another simple example of dynamic linking where `MessageBoxA` is called. The code assumes that `user32.dll`, the DLL that exports that function, isn't loaded into memory. Recall

that if a DLL isn't loaded into memory the usage of `LoadLibrary` is required to load that DLL into the process's address space.

```
typedef int (WINAPI* MessageBoxAFunctionPointer)( // Constructing a new
data type, that will represent MessageBoxA's function pointer
  HWND           hWnd,
  LPCSTR         lpText,
  LPCSTR         lpCaption,
  UINT           uType
);

void call(){
    // Retrieving MessageBox's address, and saving it to 'pMessageBoxA'
(MessageBoxA's function pointer)
    MessageBoxAFunctionPointer pMessageBoxA =
(MessageBoxAFunctionPointer)GetProcAddress(LoadLibraryA("user32.dll"),
"MessageBoxA");
    if (pMessageBoxA != NULL){
        // Calling MessageBox via its function pointer if not null
        pMessageBoxA(NULL, "MessageBox's Text", "MessageBox's Caption",
MB_OK);
    }
}
```

Function Pointers

For the remainder of the course, the function pointer data types will have a naming convention that uses the WinAPI's name prefixed with `fn`, which stands for "function pointer". For example, the above `MessageBoxAFunctionPointer` data type will be represented as `fnMessageBoxA`. This is used to maintain simplicity and improve clarity throughout the course.

Rundll32.exe

There are a couple of ways to run exported functions without using a programmatical method. One common technique is to use the rundll32.exe binary. `Rundll32.exe` is a built-in Windows binary that is used to run an exported function of a DLL file. To run an exported function use the following command:

```
rundll32.exe <dllname>, <function exported to run>
```

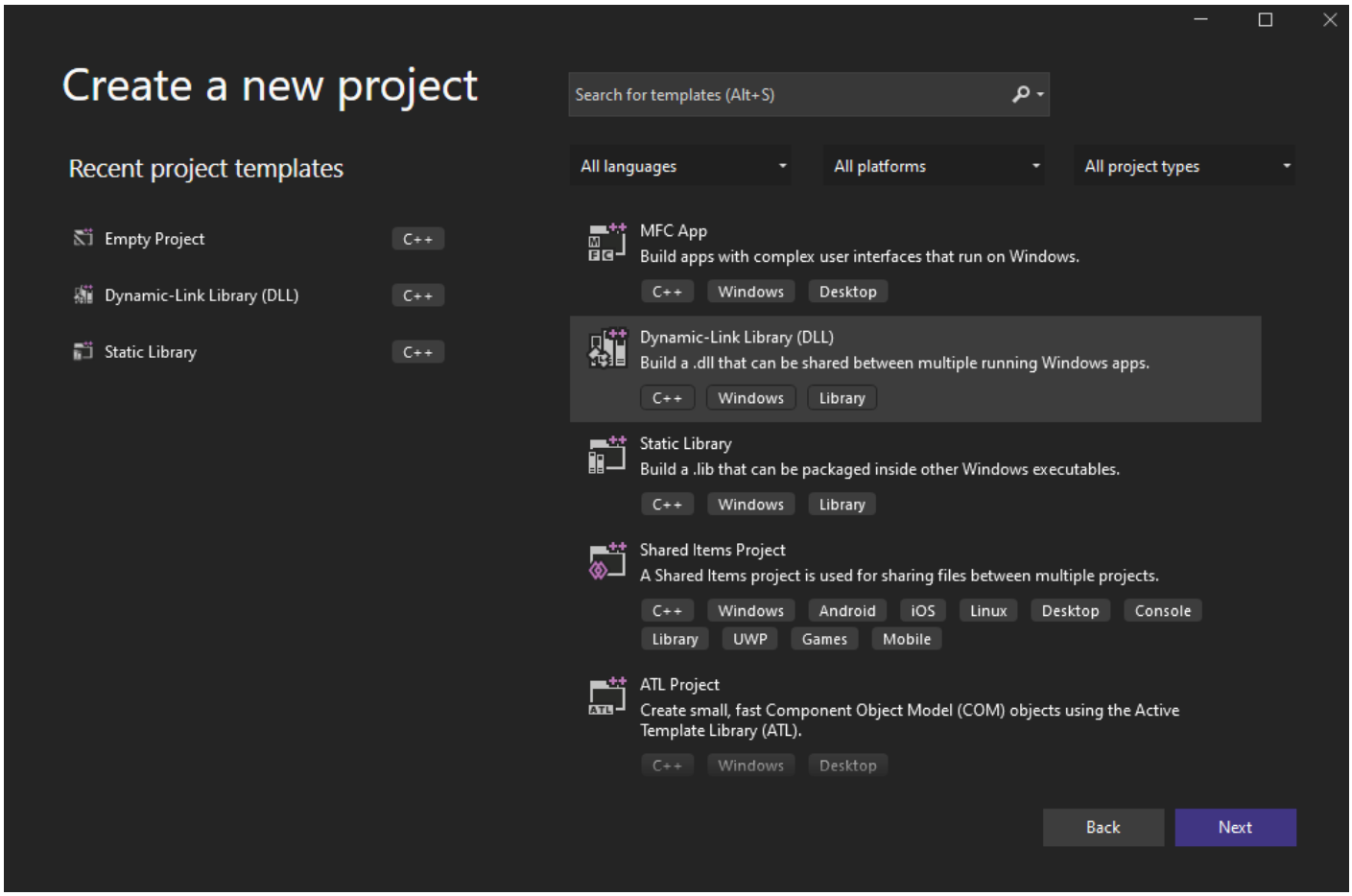
For example, `User32.dll` exports the function `LockWorkStation` which locks the machine. To run the function, use the following command:

```
rundll32.exe user32.dll,LockWorkStation
```

Removing Precompiled Headers

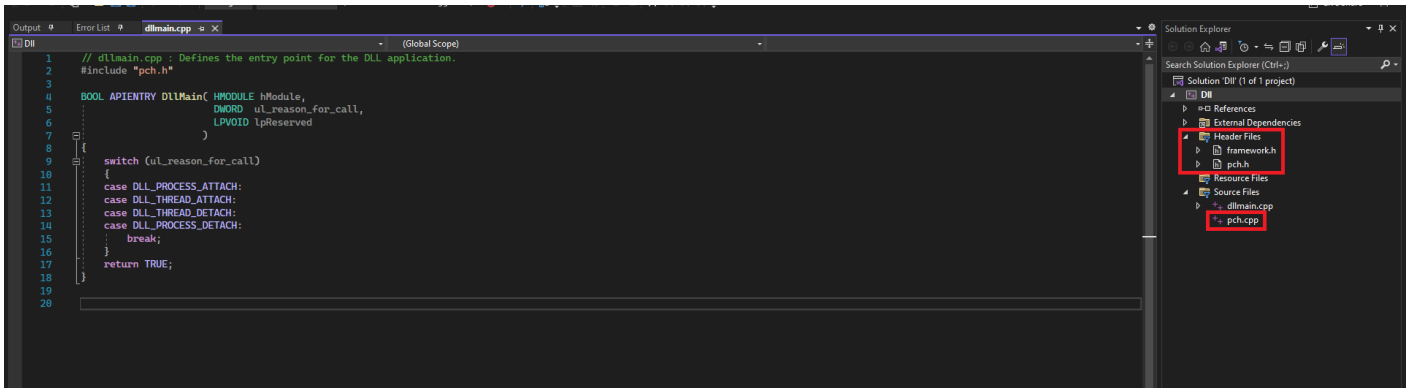
When creating a DLL file using the Visual Studio template, the DLL template will come with `framework.h`, `pch.h` and `pch.cpp` which are known as Precompiled Headers. These are files used to make the project compilation faster for large projects. It is unlikely that these will be required in this situation and therefore it is recommended to delete these files using the steps below.

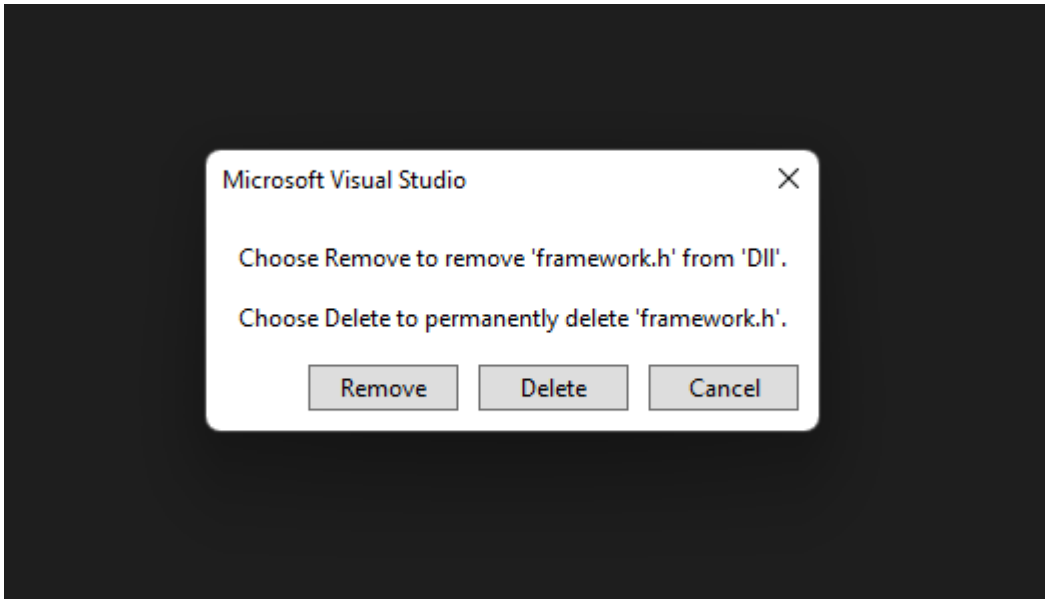
First, create a new DLL file using Visual Studio's DLL template like previously shown.



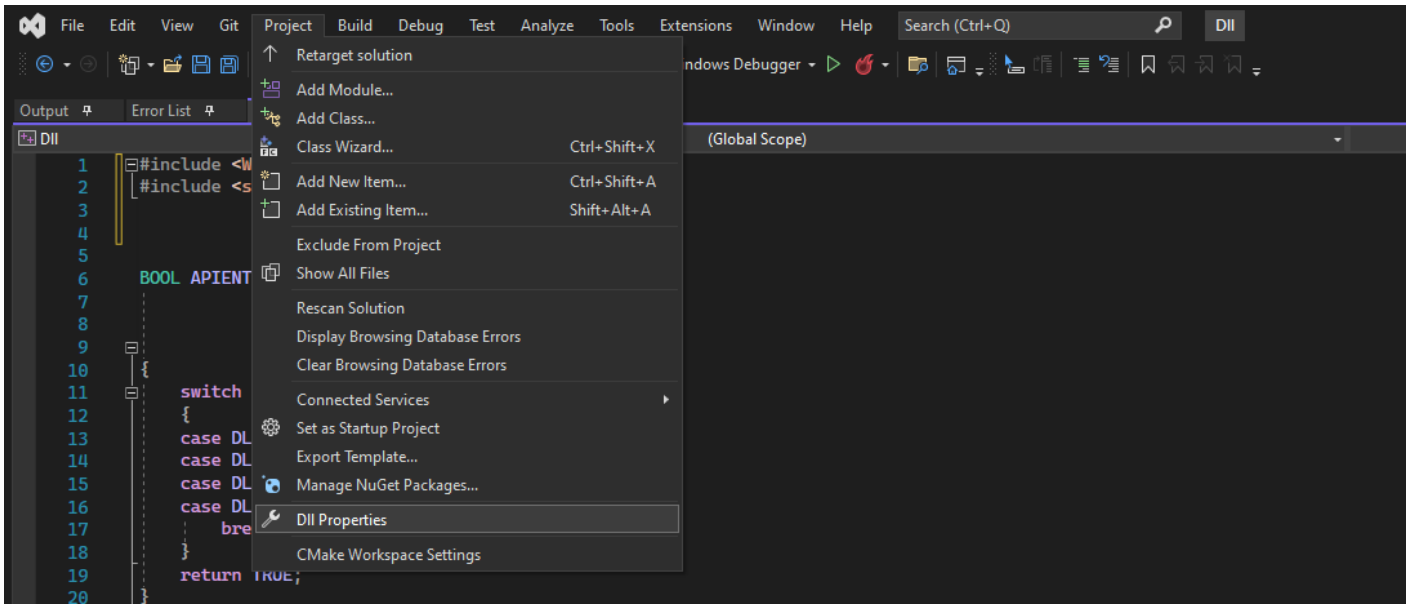
Next, open the project and highlight `framework.h`, `pch.h` and `pch.cpp` and press the delete key and select the 'Delete' option.

You will also need to remove `#include "pch.h"` from `dllmain.cpp` and replace it with `#include <Windows.h>`.

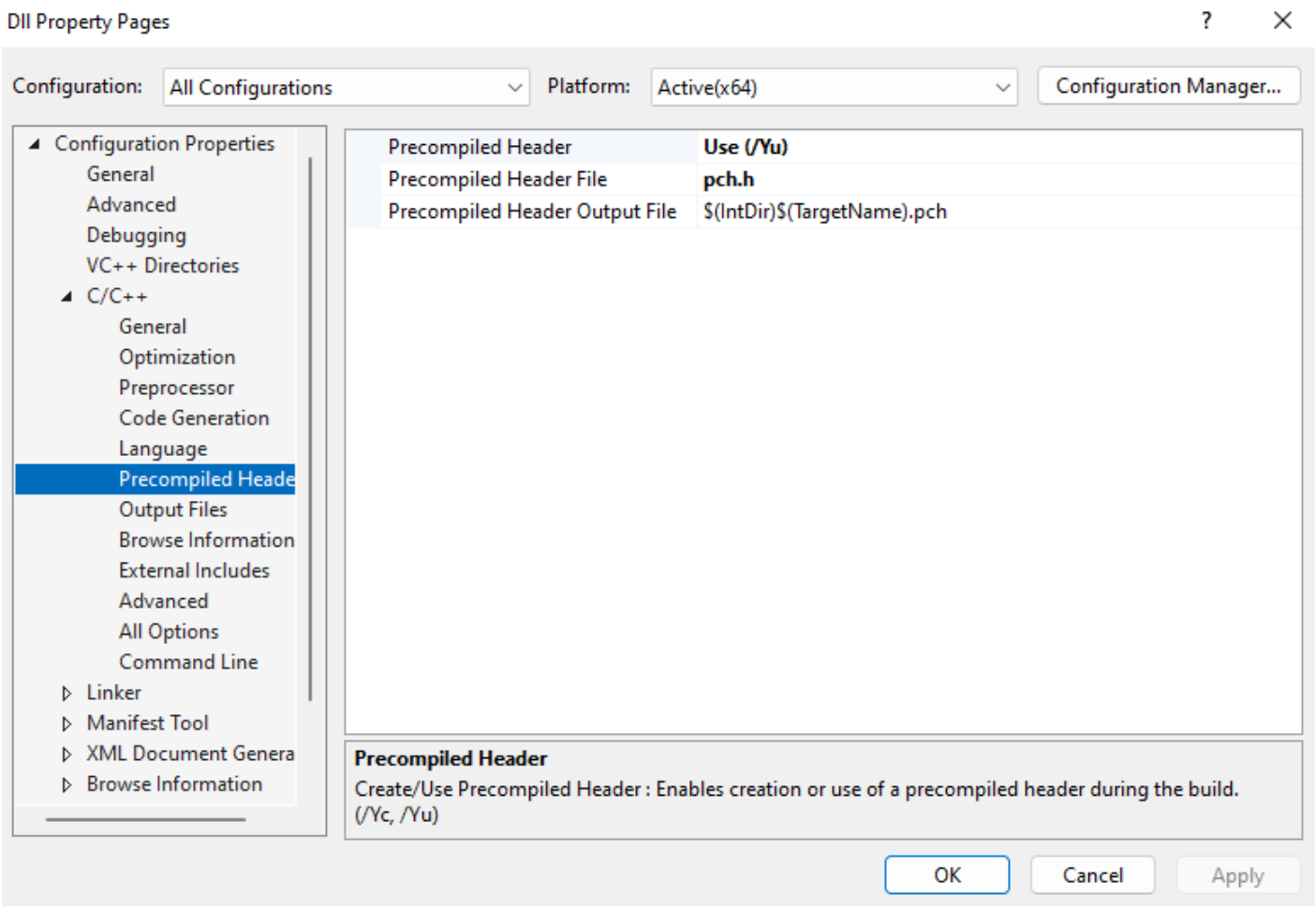




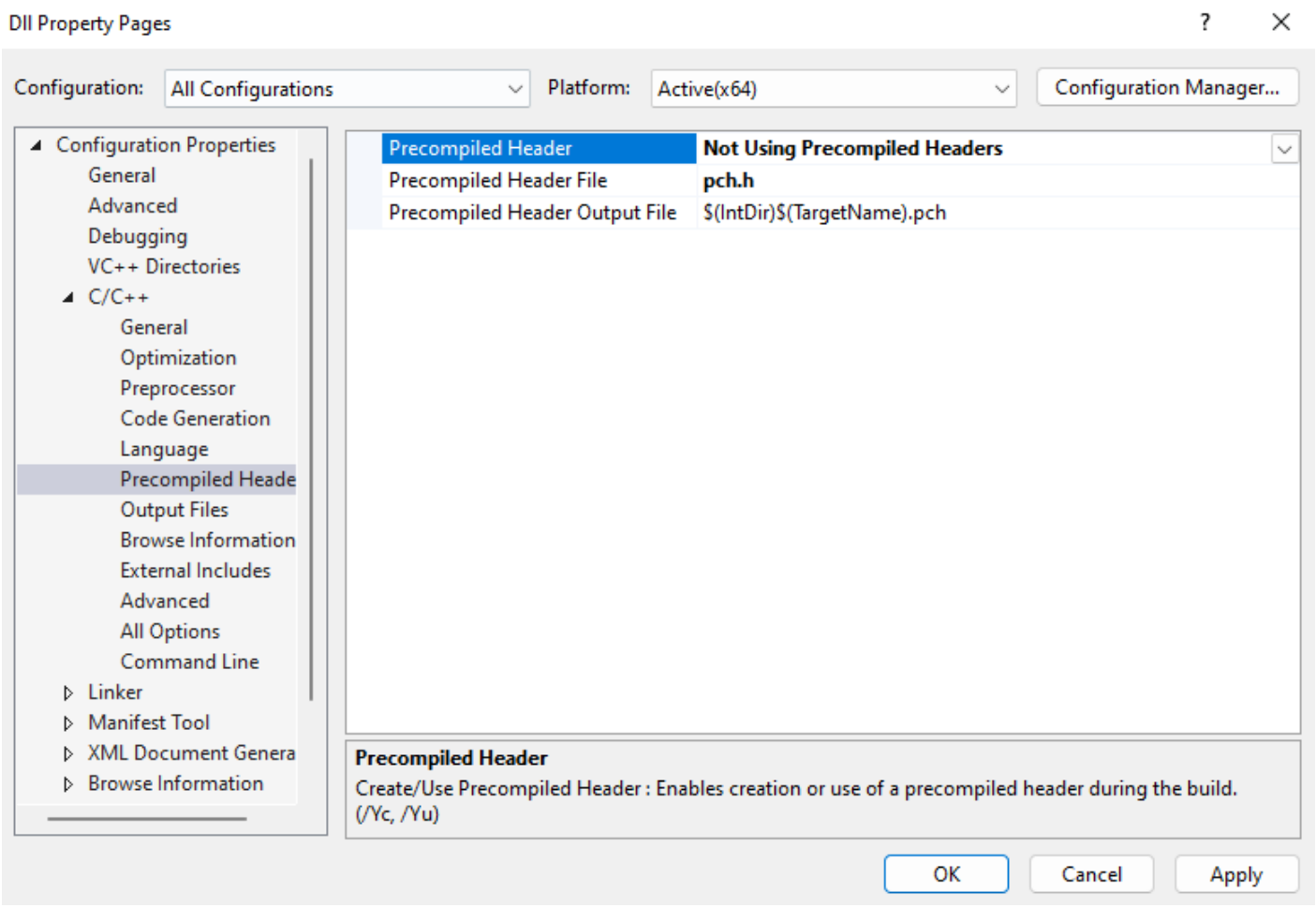
After deleting the precompiled headers, the compiler's default settings must be changed to confirm that precompiled headers should not be used in the project.



Go to **C/C++ > Precompiled Header**



Change the 'Precompiled Header' option to 'Not Using Precompiled Headers' and press 'Apply'.



Finally, change the `dllmain.cpp` file to `dllmain.c`. This is required since the provided code snippets in Maldev Academy use C instead of C++. To compile the program, click Build > Build Solution and a DLL will be created under the *Release* or *Debug* folder, depending on the compile configuration.