EEEE2076 -Software Development Group Design Project

Worksheet 3 - Software Development with Visual Studio

P Evans

January 10, 2025

Contents

1 Introduction

1.1 Aims

In the past when you have written code you have used an Integrated Development Environment (IDE) and this has managed the compile and build process. In the background, the IDE is actually issuing a series of commands to programs such as the *compiler* and *linker* to translate your source code into an executable program. This IDE approach is acceptable for simple coding exercises where a single person will write and compile the code on a single computer. For larger software engineering projects where multiple people will work on code that must compile on different machines, possibly running different operating systems, the basic IDE approach quickly becomes unmanageable.

For larger projects of this type, you must:

- Have a mechanism for centrally storing and sharing code and allowing multiple users on different machines to develop code in a collaborative manner (e.g Git)
- Have mechanisms that allow each user to easily compile the code on their machine, regardless of the particular configuration of that machine or the operating system that it is running.

In order to do this we must take control of the compile and build process but first we need to understand how that process works. We will also look at the concept of *libraries* that allow efficient reuse of compiled code and finally see how we can automate the build process in a platform independent manner using the *CMake Build System* (next worksheet).

1.2 Visual Studio

Visual Studio is Microsoft's official IDE and compiler suite for Windows. In the past you have probably used the Code::Blocks IDE which uses the MinGW compiler, MinGW is a Windows port of the GNU compiler collection which is the basis of Linux and other open-source, Unix-like operating systems. For this project we will use Visual Studio, whilst it is technically possible to use MinGW on Windows (or even develop the software for Linux or MacOS), there are a number of reasons why we won't be doing this:

- We need to use the VTK library which does not fully support MinGW on Windows. It will work, but usually you have to deal with a number of problems which cost time. It should work first-time with Visual Studio, if you follow the instructions!
- Valve's OpenVR library is supplied precompiled for Visual Studio, we will use this later.
- Compiling VTK is a computationally intensive process and Visual Studio's compiler is much more efficient than MinGW. Again, this saves you time.
- If you become a Windows software developer, you are likely to have to use Visual Studio so its good to get some experience.
- If we have people using different compilers and operating systems it becomes very difficult to support everyone properly due to the wide range of problems and compatibility issues that will occur.

2 Downloading and Installing Visual Studio

Go to the Microsoft Visual Studio website https://visualstudio.microsoft.com/vs/community/ and download Visual Studio Community Edition. Community edition is the free version. Although you do have access to the paid versions of Visual Studio for free through the university, I recommend just using Community as we don't need anything more and you don't need to worry about the license. When installing Visual Studio, make sure you select the *Desktop development with C++* option, see Fig. 1.

3 Compiling a simple application using command line tools

3.1 Compiling and Linking

A number of steps are performed when you *build* a program and for larger software projects, understanding (and being able to control) these steps is important. The two basic operations that are performed are:

- *Compiling* This process translates human readable source code for all of your functions into machine readable instructions. It can be seen as a direct translation between human language and machine language. It does not create a useful output (i.e. an executable program).

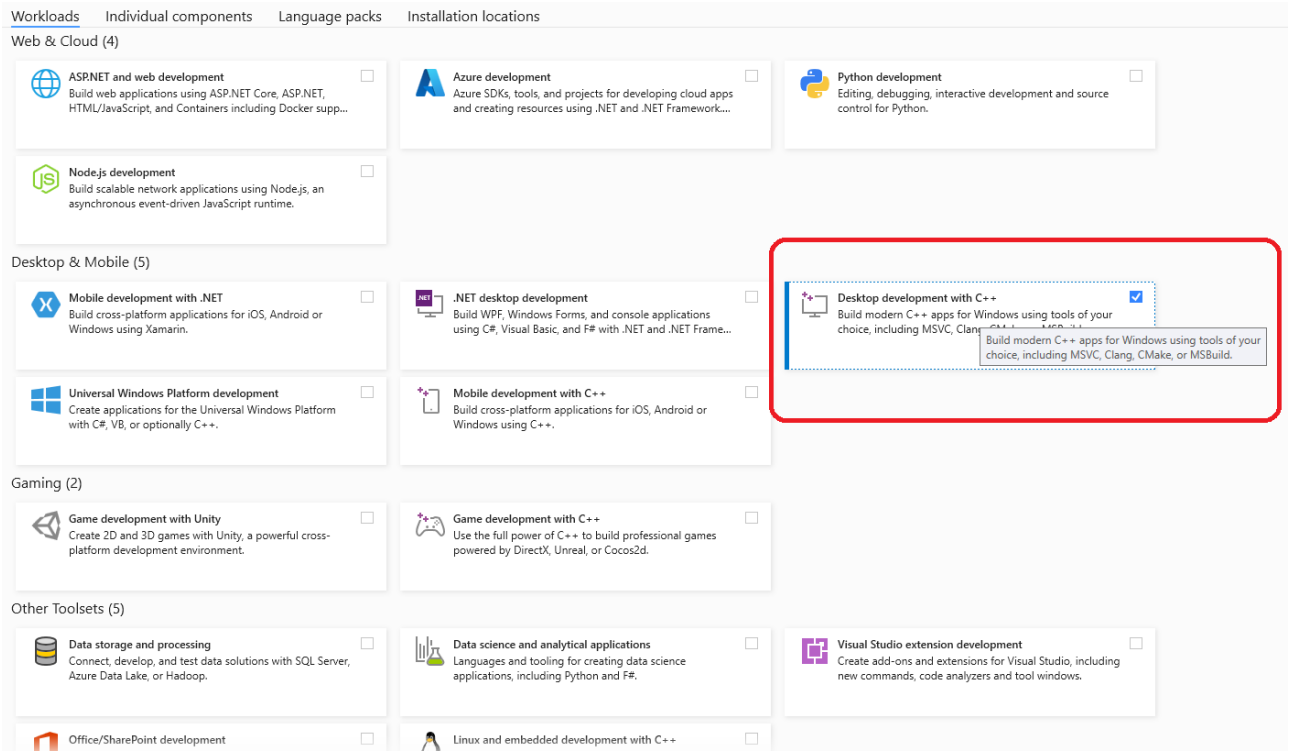

Figure 1: Installing Visual Studio - make sure you select the Desktop development with C++ option!

- *Linking* This process takes the machine readable instructions for all of your functions and links them into a single, coherent block of machine code. It then adds additional data to turn this machine code into a useful output (the executable program).

The following example will show how these steps are performed in the background when you click *Build* in an IDE.

3.1.1 Exercise 1 - Setting Up

Create a new folder in your individual Github repository. You will use this for the following exercises. Remember to regularly add, commit and push your work, ideally after every exercise. If you accidentally delete something or you have technology problems later in the semester the work will always available in Github.

Now you need to initialise the Visual Studio command line build environment. Open a terminal window and cd to your worksheet folder. The Studio compiler tool is called *cl*, if you type *cl* now you will almost certainly get an error saying the program cannot be found.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl
'cl' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

Visual Studio provides a batch script that can be run to setup the path and other environment variables. It can usually be found at:

C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Auxiliary\Build\vcvarsall.bat
Obviously, the exact path may be different if you have a different version of Visual Studio or chose a different install location. Once you have located the batch file, run it from your terminal. As the path contains spaces, you will need to enclose it in quotation marks. You also need to tell it that you want to build a 64 bit program by adding *x64* as an argument. Try to run *cl* again to make sure it has worked.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>"C:\Program Files\Microsoft Visual Studio\2022\
    Community\VC\Auxiliary\Build\vcvarsall.bat" x64
**********************************************************************
** Visual Studio 2022 Developer Command Prompt v17.2.1
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

C:\Users\ezzpe\2023_ezzpe\Worksheet3>cc
'cc' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

3.1.2 Exercise 2 - Building an executable from the command line

Go to the working directory for your Worksheet3 work in your individual repository and create a C++ source file, containing the following code in your working directory.

```
// begin - hello.cpp ----------
#include <iostream>
int main( int argc, char *argv[] ) {
   std::cout << "Hello Planet Earth!" << std::endl;
   return 0;
}
// end - hello.cpp ------------
```

Compile the code using cl. Note that '/c' tells cl to *compile* hello.cpp only and don't try to automatically *link* which is possible for this simple example):

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /c hello.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.cpp
D:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.32.31326\
    include\ostream(770): warning C4530: C++ exception handler used, but unwind
    semantics are not enabled. Specify /EHsc
```

```
hello.cpp(4): note: see reference to function template instantiation '
    std::basic_ostream<char,std::char_traits<char>> &std::operator <<<std::char_traits<
    char>>(std::basic_ostream<char,std::char_traits<char>> &,const char *)' being
    compiled

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

It is suggesting you use an additional option (/EHsc) when compiling, this isn't essential you can if you like:

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /c /EHsc hello.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.cpp

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

Now look at what is in the directory:

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  14:07    <DIR>          .
08/09/2023  14:07    <DIR>          ..
08/09/2023  14:06               191 hello.cpp
08/09/2023  14:11           178,752 hello.obj
2 File(s)        178,943 bytes
2 Dir(s)  34,758,213,632 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

You can see that an object file (hello.obj) has been created. This is the compiled version of your code, it is not an executable. To make an executable we must get Visual Studio to link the code. This is done with the linker program, *link*. You must tell link which files you want linking (hello.obj), and it can be good idea to tell it what to call the output using the /out:<filename> option.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>link hello.obj /out:hello.exe
Microsoft (R) Incremental Linker Version 14.32.31329.0
Copyright (C) Microsoft Corporation.  All rights reserved.


C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  14:14    <DIR>          .
08/09/2023  14:14    <DIR>          ..
08/09/2023  14:06               191 hello.cpp
08/09/2023  14:14           231,936 hello.exe
08/09/2023  14:11           178,752 hello.obj
3 File(s)        410,879 bytes
```

```
2 Dir(s)  34,863,955,968 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>l
```

By analysing the directory contents again, you can see that we now have hello.exe. The linker has taken the code generated for all of your functions and combined or linked it (ok so there was only one function so maybe it didn't have much work to do!) and added additional data to produce an executable output. Notice the difference in size between the object (a compiled version of main(), 170kB) and the final executable (230kB). There are a few possible reasons for this, including:

- There is a a lot of standard data and code that must be present in all executables, see: https://en.wikipedia.org/wiki/Portable_Executable for an overview of the Windows executable format. This is beyond the scope of this module but information includes tables that enable the program to load and find functions in *dynamic libraries.*
- The compiler may (in some cases) have statically linked your code with some pre-compiled object code for other functions you have used ('printf' or 'std::cout' for example) which it obtained from a *static libraries.*.

Make sure you add your source files to Github. Make sure you do not add any binary files (.exe, .obj, etc)!. You should be committing after each exercise. Remember your will need to add (e.g. `git add "Worksheet 3"`), commit, push (not the need for quotes if the filename has a space in it). Alternatively, if you are in the Worksheet 3 folder, the command `git add .` will add everything in the current folder.

3.1.3 Exercise 3 - Multiple Source Files and Multiple Objects

The example in the previous section was extremely simple, in reality you would usually have multiple source files, which will be compiled into multiple objects and subsequently linked into the final output executable. This example demonstrates how you can do this with cc/link. Firstly create some new source files, *calc.cpp* which will be the main program code and a second file source file *adder.cpp* which will contain a function that will be called from within *calc.cpp*. *adder.cpp* will also need its own header file *adder.h*:

```cpp
//begin - calc.cpp ----------
#include <sstream>
#include <iostream>

int main( int argc, char *argv[] ) {
   int a, b, c;

   if(argc != 3) return 1;

   std::string sa( argv[1] );
   std::string sb( argv[2] );

   std::stringstream ssa( sa );
   std::stringstream ssb( sb );

   ssa >> a;
   ssb >> b;

   c = add( a, b );
   std::cout << c;

   return 0;
}
```

```
//end - calc.cpp ----------
```

```
//begin - adder.cpp ----------
#include "adder.h"
int add( int a, int b ) {
   return a + b;
}
//end - adder.cpp ----------
```

```
//begin - adder.h ----------
// The following two lines prevent adder.h from being included
// more than once in any source (.cpp) file. If this were to happen
// it would cause problems in the compile process but it is difficult to
// prevent manually in large projects. These #ifndef, #define and #endif
// commands for an "include guard" and are types of compiler directive.
// The include guard contains an ID for this file "MATHSLIB_ADDER_H", this is
// arbitrary but must be unique within the project.

#ifndef MATHSLIB_ADDER_H
   #define MATHSLIB_ADDER_H

   // prototype for our function
   int add( int a, int b );

#endif
//end - adder.h ----------
```

Now compile calc.c to get an object file:

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /c /EHsc calc.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

calc.cpp
calc.cpp(19): error C3861: 'add': identifier not found

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

There was an error, why is this? Work out what the compiler error means and fix the error by modifying calc.cpp. Then compile calc.cpp and adder.cpp.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  15:51    <DIR>          .
08/09/2023  15:51    <DIR>          ..
08/09/2023  15:43               165 adder.cpp
08/09/2023  15:50               714 adder.h
08/09/2023  15:51               642 adder.obj
08/09/2023  15:50               462 calc.cpp
08/09/2023  15:51           468,399 calc.obj
08/09/2023  15:46           231,936 helldir
```

```
08/09/2023  14:06               191 hello.cpp
7 File(s)        702,509 bytes
2 Dir(s)  34,919,620,608 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

It should be obvious that we now have object files for both calc.cpp and adder.cpp. Use *link* to link them and build an executable and test it.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>link adder.obj calc.obj /out:calc.exe
Microsoft (R) Incremental Linker Version 14.32.31329.0
Copyright (C) Microsoft Corporation.  All rights reserved.


C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  15:53    <DIR>          .
08/09/2023  15:53    <DIR>          ..
08/09/2023  15:43               165 adder.cpp
08/09/2023  15:50               714 adder.h
08/09/2023  15:51               642 adder.obj
08/09/2023  15:50               462 calc.cpp
08/09/2023  15:53           326,656 calc.exe
08/09/2023  15:51           468,399 calc.obj
08/09/2023  15:46           231,936 helldir
08/09/2023  14:06               191 hello.cpp
8 File(s)      1,029,165 bytes
2 Dir(s)  34,882,879,488 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>calc 6 7
13
C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

This simple example has shown that producing software from source code is a two stage process:

Source Code ->Compile ->Object Files (Machine Code) ->Link ->Output (Executable or library)

Crucially, the compilation process for each source file is independent of the final linking process for the executable. So what if some of our source files are reused in multiple projects? For example the adder code may be used in the calc program, it could be used in a simple game, etc. Wouldn't it be more efficient if the adder code could be compiled only once, the compiled code saved and simply linked into new projects when needed? For the adder code there would be little benefit since compiling 3 lines of code is not a significant overhead, but imagine if we had 20,000 lines of code designed to solve partial differential equations. Recompiling this whenever it is reused in a project would be very inefficient. For this reason, it is common to package code that may be reused into *libraries*.

4 Libraries

4.1 What are they?

A library that is just some precompiled code that may be reused by many software projects, think of it as a reusable object file. There are two types of library:

- *Static Libraries* are linked to each project by the linker when the project is built. A copy of the library's 'object' code is inserted into the executable that is being built during the linking process.
- *Dynamic libraries* or *Shared libraries* are linked with the executable only when it is executed (at runtime). The linker will NOT include a copy of the library's object code in the executable being built, the executable is dependent on being able to find a copy of this code on the operating system when it runs. The linking happens dynamically at runtime.

We will first look at static libraries as these are quite simple and then take a more detailed look why dynamic libraries are used and how we can develop software that uses them.

4.2 Static Libraries

Static libraries are a form of object file (*.obj*) that can be easily reused in multiple software projects. Visual Studio gives static libraries the *.lib* file extension. A static library is simply an archive containing 1 or more object files. A static library containing only 1 object file is a little bit pointless, but for simplicity that's where we'll start!

4.2.1 Exercise 4 - Create and use a Static Library

The process for creating a static library involves two steps: create the object file, then convert this to a static library using the *lib* tool.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  16:15    <DIR>          .
08/09/2023  16:15    <DIR>          ..
08/09/2023  15:43               165 adder.cpp
08/09/2023  15:50               714 adder.h
08/09/2023  15:50               462 calc.cpp
08/09/2023  14:06               191 hello.cpp
4 File(s)          1,532 bytes
2 Dir(s)  34,827,227,136 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /c adder.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

adder.cpp

C:\Users\ezzpe\2023_ezzpe\Worksheet3>lib adder.obj /out:libadder.lib
Microsoft (R) Library Manager Version 14.32.31329.0
Copyright (C) Microsoft Corporation.  All rights reserved.
```

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  16:15    <DIR>          .
08/09/2023  16:15    <DIR>          ..
08/09/2023  15:43               165 adder.cpp
08/09/2023  15:50               714 adder.h
08/09/2023  16:15               880 libadder.lib
08/09/2023  16:15               642 adder.obj
08/09/2023  15:50               462 calc.cpp
08/09/2023  14:06               191 hello.cpp
6 File(s)          3,054 bytes
2 Dir(s)  34,811,072,512 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

To complete the process, create the calc object and link as before, but now link against the static library (*.lib*) rather than the object.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /c /EHsc calc.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

calc.cpp

C:\Users\ezzpe\2023_ezzpe\Worksheet3>link calc.obj libadder.lib /out:calc_static.exe
Microsoft (R) Incremental Linker Version 14.32.31329.0
Copyright (C) Microsoft Corporation.  All rights reserved.


C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  16:47    <DIR>          .
08/09/2023  16:47    <DIR>          ..
08/09/2023  15:43               165 adder.cpp
08/09/2023  15:50               714 adder.h
08/09/2023  16:15               880 libadder.lib
08/09/2023  16:15               642 adder.obj
08/09/2023  15:50               462 calc.cpp
08/09/2023  16:46           468,399 calc.obj
08/09/2023  16:47           326,656 calc_static.exe
08/09/2023  14:06               191 hello.cpp
8 File(s)        798,109 bytes
2 Dir(s)  34,721,189,888 bytes free

```

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

In real applications (e.g. VTK) libraries are usually stored in a common *libraries* or *lib* directory somewhere else on the file system, this means that you must tell the linker where to find the library when building your application. The header files associated with the library are usually stored in an *include* directory. The following example will create a typical library structure using the files you have just built.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>mkdir adder_static

C:\Users\ezzpe\2023_ezzpe\Worksheet3>mkdir adder_static\lib

C:\Users\ezzpe\2023_ezzpe\Worksheet3>move libadder.lib adder_static\lib
1 file(s) moved.

C:\Users\ezzpe\2023_ezzpe\Worksheet3>mkdir adder_static\include

C:\Users\ezzpe\2023_ezzpe\Worksheet3>move adder.h adder_static\include
1 file(s) moved.

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

We'll now delete the old .exe and create a new one, linking against the adder library which is now in another directory:

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>del *.exe *.obj

C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /c calc.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

calc.cpp
calc.cpp(5): fatal error C1083: Cannot open include file: 'adder.h': No such file or
    directory

C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /EHsc /c calc.cpp /Iadder_static/include
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

calc.cpp
```

```

C:\Users\ezzpe\2023_ezzpe\Worksheet3>link calc.obj libadder.lib /out:calc_static.exe
Microsoft (R) Incremental Linker Version 14.32.31329.0
Copyright (C) Microsoft Corporation.  All rights reserved.

LINK : fatal error LNK1181: cannot open input file 'libadder.lib'

C:\Users\ezzpe\2023_ezzpe\Worksheet3>link calc.obj libadder.lib /out:calc_static.exe
    /libpath:adder_static/lib
Microsoft (R) Incremental Linker Version 14.32.31329.0
Copyright (C) Microsoft Corporation.  All rights reserved.


C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
```

```
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  16:59    <DIR>          .
08/09/2023  16:59    <DIR>          ..
08/09/2023  15:43               165 adder.cpp
08/09/2023  16:54    <DIR>          adder_static
08/09/2023  15:50               462 calc.cpp
08/09/2023  16:59           326,656 calc_static.exe
08/09/2023  16:58           468,399 calc.obj
08/09/2023  14:06               191 hello.cpp
5 File(s)        795,873 bytes
3 Dir(s)  34,873,774,080 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

Commit your code changes, create a zip of your static library and executable and upload this as a release to Github

There are a few key differences when dealing with libraries that are located in a different directory to the build directory:

- When compiling the object files, we must tell the compiler where to find header files that are included in our source files. We do this with the compiler option */Ipath_to_header_directory* (where *path_to_header_directory* is the path to the directory containing the .h files). You can have more than one /I option if header files are in multiple locations. (Note: I in /I is a upper case i)
- When linking our program, we must tell the linker where to find our libraries, We do this using the */libpath:path_to_lib_directory* linker option (where *path_to_lib_directory* is the path to the directory containing the .lib files). You can have more than one /libpath option if libraries are in multiple locations.

You can probably understand that for projects which depend on lots of static libraries, the commands issued to the compiler and linker can become incredibly long and complicated. Because of this, it is common to use a *build system generator* to automatically determine which libraries we need and where they are located, and then a *build system* to issue the correct commands to the compiler and linker. We will cover this later.

4.3 Dynamic Libraries

Static libraries make compilation more efficient but use of static libraries can make the executables produced larger than they need to be. As an example:

Most programs that run on the Windows operating system and interact with the Windows GUI will directly or indirectly utilise a set of common functions, defined in the Windows API. These functions allow Windows applications to create windows, menus etc and capture user input. Microsoft provides these functions as a set of libraries for developers to use. If these were static libraries, almost every single program running on Windows would contain a physical copy of some of these functions. This would mean that exactly the same code, for the same function, could exist hundreds of times in different executables across the system. This would be an inefficient use of disk storage space, and also RAM if many of these programs were running at the same time.

The solution to this problem is the *dynamic library*, *shared library* or *shared object*. A single copy of the library exists on the computer and applications dynamically link to, and share, this library when they run. On Windows these libraries are called Dynamic Link Libraries (*.dll*), on Linux they are called Shared Objects (*.so*), and on MacOS Dynamic Libraries (*.dylib*).

The following example will create a dynamic version of the maths library and a new version of calc (calc_dynamic.exe) that utilises it.

4.3.1 Exercise 5 - Create and use a dynamic library

First we will create a new library header file (adder.h) that is compatible with a dynamic library. The reasons for these modifications are explained below below. Create anew adder.h in the Worksheet3 folder with the following code.

```
//begin - adder.h ----------------

// The following two lines prevent adder.h from being included
// more than once in any source (.cpp) file. If this were to happen
// it would cause problems in the compile process but it is difficult to
// prevent manually in large projects. These #ifndef, #define and #endif
// commands for an "include guard" and are types of compiler directive.
// The include guard contains an ID for this file "MATHSLIB_ADDER_H", this is
// arbitrary but must be unique within the project.
#ifndef MATHSLIB_ADDER_H
   #define MATHSLIB_ADDER_H

   // We need to prefix our function names with an additional
   // keyword which is different depending on the operating
   // system we are using, and whether we are using or creating the
   // library.
   // The variables "maths_EXPORTS" must be defined at build time if
   // we are building the library, but not if we are using it.
   #if defined(WIN32) | defined(_WIN32)
      #ifdef maths_STATIC
         // dont add any keywords if building a static library
         #define MATHSLIB_API
      #else
         #ifdef maths_EXPORTS
            #define MATHSLIB_API __declspec( dllexport )
         #else
            #define MATHSLIB_API __declspec( dllimport )
         #endif
      #endif
   #else
      // MATHSLIB_API is defined as nothing if were not on Windows
      #define MATHSLIB_API
   #endif

   // The above will include one of:
   // __declspec( dllexport )
   // __declspec( dllimport )
   // before declarations. This is a Microsoft specific
   // extension to C/C++

   // prototype for the function including additional keyword
   MATHSLIB_API int add( int a, int b );
#endif

//end - adder.h ----------------
```

Now build the dynamic library.

```
C:\Users\ezzpe\2023_ezzpe\Worksheet3>cl /c /Dmaths_EXPORTS adder.cpp
```

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31329 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

adder.cpp

C:\Users\ezzpe\2023_ezzpe\Worksheet3>link /dll adder.obj /out:libadder.dll
Microsoft (R) Incremental Linker Version 14.32.31329.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Creating library libadder.lib and object libadder.exp

C:\Users\ezzpe\2023_ezzpe\Worksheet3>dir
Volume in drive C is Windows
Volume Serial Number is BEEA-C413

Directory of C:\Users\ezzpe\2023_ezzpe\Worksheet3

08/09/2023  17:40    <DIR>          .
08/09/2023  17:40    <DIR>          ..
08/09/2023  15:43               165 adder.cpp
08/09/2023  17:40             1,640 adder.h
08/09/2023  17:40               664 adder.obj
08/09/2023  16:54    <DIR>          adder_static
08/09/2023  15:50               462 calc.cpp
08/09/2023  14:06               191 hello.cpp
08/09/2023  17:40            91,648 libadder.dll
08/09/2023  17:40               694 libadder.exp
08/09/2023  17:40             1,740 libadder.lib
8 File(s)         97,204 bytes
3 Dir(s)  34,821,922,816 bytes free

C:\Users\ezzpe\2023_ezzpe\Worksheet3>
```

- the */Dmaths_EXPORTS* option defines the maths_EXPORTS variable for adder.h to use. In turn, this causes the add function to be exported (i.e. it is made available in the DLL for other programs to use)
- the */dll* option passed to *link* tells it to create a shared library.
- The three outputs from this process are:
  - *libadder.dll* This is our dynamic/shared library and contains the adder code. It will be loaded/linked by calc_dynamic.exe at run-time.
  - *libadder.lib* This is a small static import library that we must link to calc_dynamic.exe at compile-time. This small static library contains the instructions that will tell calc_dynamic.exe how to load libadder.dll at runtime.
  - *libadder.exp* This is an index file describing the functions exported by the DLL and is used by the linker in some circumstances.

To finish this worksheet - create a *adder_dynamic* directory with lib and include subdirectories and copy the dynamic library into it. Create a calc_dynamic.exe and link against this library. Make sure all of your code is committed to Github, and upload the library and executable as a release.

Make sure your program actually works - the .exe will need to find the .dll. This means that the dll either needs to be in the same directory as the .exe, or if not, the folder with the DLL must be added to your PATH variable.

5 Notes on Type Modifiers (declspec() etc) and Calling Conventions

declspec(dllimport) and *declspec(dllexport)* are Windows specific type modifiers. When object code is produced, the name of functions is encoded into the object file (and therefore dynamic libraries that result from this object file) in a specific format. When your program loads the dynamic library at run-time, it needs to be able to locate functions in it using their name. Obviously the calling program (calc_dynamic.exe) needs to have the same expectations about how the function names should be encoded into dynamic libraries as the compiler that created the dynamic libraries. Remember, there's no guarantee that the .dll and .exe will have been created on exactly the same machine / compiler. These modifiers are an instruction to the compiler (of either the .exe or .dll) of how function names should be encoded. See here.

There is also a related topic of how function calls are implemented in machine code, *Calling Conventions*, that you may find interesting See here.

6 Analysing Dependencies

On Windows you can download a program called dependencies which will give you a dependency tree that illustrates which dynamic libraries a program depends on and also identify compatibility issues (32bit program trying to load a 64bit library for instance). Download it and try on calc_dynamic.exe.