H62PEP Computing - Worksheet 3 - Compilers and Building Software from the Command Line

P. Evans

A quick recap and the motivation for worksheet

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 (**multiple platforms**), 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).

Preparation

To go any further, you need a compiler! You have some choices, which include:

- GCC - The GNU Compiler Collection (gcc) is at the core of GNU operating systems (i.e. Linux). There a few groups who have *ported* both 32 and 64bit version of this compiler to windows also. The C++ compiler program on Windows is `g++.exe`.

- Visual Studio. If you have Visual Studio installed, or can install it, you can use Microsoft's compiler. On Windows, the compiler program is called `cl.exe` but you must first run a batch file (e.g. `"C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat" amd64`) which will add cl.exe and other relevant utilities to the system path.
- Clang - A compiler developed by Apple, the compiler program is called `clang`.

The following exercises assume you are using Windows as that's whats available in the lab. If you are running Windows use MinGW to begin with, *we recommend downloading MinGW as part of Qt package as it will make your life easier in the next project week.* The final exercise will ask you to repeat some of these exercises using Visual Studio so you are aware of the differences.

If you are not using Windows just use the compiler that comes with your OS, don't worry about repeating the exercises on another compiler as you will already have had to resolve some issues!

There are 6 individual exercises that allow you to learn about compilers and how to use them on the command line. ***You should backup your code to your individual Github repository*** so that you have a record of your progress, you will also need some of the files your create for Worksheet4. ***The individual repository must be visible to me (`plevans`) and will be checked at the end of the semester. Completion of these exercises counts towards your individual contribution assessment***

Exercise 1 - Download a Compiler

If using Windows:

- Download and install the MinGW compiler for Windows. There are a number of different packages that you can download for this including Standard 32bit MingW, Mingw-w64, as part of the MSys platform, or as part of the Qt software development kits. We are going to use Qt in the next project week so we suggest downloading MinGW as part of Qt as this will guarantee that you get a compatbile compiler version. If you are not sure how to download Qt and MinGW, see the instructions at the beginning of Worksheet5.
- We will use the Windows command prompt (`cmd.exe`) to compile the examples. You need to make sure that the directory containing `g++.exe` is present the PATH environment variable if it isn't already. Qt will install MinGW in a folder such as `C:\Qt\Tools\mingw530_32`. You need to make sure MingW's binary directory (`C:\Qt\Tools\mingw530_32\bin\bin`) is in the path variable as this is where the compiler executables are located.
- Create a working directory on your individual `repo` called `Week2`, open `cmd.exe` and cd to this directory. Check you can run `g++.exe`. If not, fix the problem.
- Type '`g++ --help`', this gives you a list of options that can be passed to g++. Try to understand what they mean, we will explore some of these in the following exercises.

for other OSs: * Find out which compiler you have installed and how to invoke it to view the help information, and to compile a C++ source file. Ask if you are unsure how to proceed. If you have g++ (you definitely will if you're using Linux) then the steps below for Windows will be almost identical for you, they are likely to be very similar even if you are using another compiler on an Apple.

Compiling an Executable - The Build Process

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:

1. **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).
2. **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).

For simple programs g++ can actually perform both of these steps in a single operation, but we can manually break the process down to understand what is going on. This is important in understanding the build process for larger software projects.

Exercise 2 - Building an Executable from the Command Line

- Create a working directory for your Worksheet 3 work in your individual repository.
- Now create a C++ source file, containing the following code in your working directory.

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

- Compile the code using g++. Note that `-c` tells g++ to *compile* hello.cpp only:

```
>g++ -c hello.cpp
```

- Now look at the contents of the directory:

```
>dir
    10/08/2017  10:56    <DIR>          .
    10/08/2017  10:56    <DIR>          ..
    10/08/2017  10:54               129 hello.cpp
    10/08/2017  10:56             1,759 hello.o
                2 File(s)          1,922 bytes
                2 Dir(s)  92,719,759,360 bytes free
>
```

- You can see that an object file (hello.o) has been created. This is the compiled version of your code, it is not an executable. To make an executable we must get g++ to link the code. Note that the `-o` option tells g++ to create an executable output called hello.exe from the object files listed:

```
>g++ hello.o -o hello.exe
>dir
    10/08/2017  10:56    <DIR>          .
    10/08/2017  10:56    <DIR>          ..
    10/08/2017  10:54               129 hello.cpp
    10/08/2017  10:57                        70,402 hello.exe
    10/08/2017  10:56             1,759 hello.o
    3 File(s)         72,403 bytes
    2 Dir(s)  92,719,759,360 bytes free
>
```

- By analysing the directory contents again, you can see that we now have hello.exe. The linker within g++ 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(), ~1kB) and the final executable (~70kB). 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 here for an overview of the Windows executable format, although beyond the scope of this module!)
  - The compiler may (in some cases) have 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 **library**. More on this later...

Compiling an Executable 2 - 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 g++.

Exercise 3 - Compilation from Multiple Source Files

- 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*:

```
// 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;
}
// /calc.cpp ---------------------------------
```

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

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

- Now compile calc.c to get an object file:

```
>g++ -c calc.cpp
    calc.cpp: In function 'int main(int, char**)':
    calc.cpp:18:14: error: 'add' was not declared in this scope
    c = add(a, b);
```

- There was an error, why is this? Fix the error by modifying calc.cpp. Then compile calc.cpp, then adder.cpp.

```
>g++ -c calc.cpp
>g++ -c adder.cpp
>dir
    10/08/2017  12:45    <DIR>          .
    10/08/2017  12:45    <DIR>          ..
    10/08/2017  12:32                66 adder.cpp
    10/08/2017  12:45               575 adder.h
    10/08/2017  12:45               630 adder.o
    10/08/2017  12:44               350 calc.cpp
    10/08/2017  12:45             3,172 calc.o
    10/08/2017  12:44               128 hello.cpp
    10/08/2017  11:26            70,402 hello.exe
    10/08/2017  11:26             1,714 hello.o
8 File(s)         77,037 bytes
2 Dir(s)  92,675,997,696 bytes free
```

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

```
>g++ calc.o adder.o -o calc.exe
>calc 6 7
    13
```

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*.

Libraries

What are they?

In simple terms 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 g++ linker when the project is built. A copy of the library's 'object' code is inserted into the executable that is being built by g++ during the linking process.
- **Dynamic libraries** or **Shared libraries** are linked with the executable only when it is executed (at runtime). g++ 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.

Static Libraries

Static libraries are a form of object file (*.o*) that can be easily reused in multiple software projects. g++ gives static libraries the *.a* file extension and by convention prefixes the library name with *lib*, e.g. *libmaths.a* is a valid static library name. So, how do we create static libraries, how do we use them, and how are they different from object files?

What is a static library?

A static library is simply an *archive* containing 1 or more object (.o) files. A static library containing only 1 object file is a little bit pointless, but for simplicity that's where we'll start!

Exercise 4 - Create and Use a Simple Library

We'll use the calc-adder example. adder.cpp will be placed in a static library called libmaths.a and we'll then create and executable that uses this library. * First, to create the static library, we need to compile adder.cpp to an object file and then create the static library from this object. The ar (archiver) program is used to convert 1 or more objects into a static library. For information on how to use `ar`, and what the command line options mean, type `ar --help` in the command prompt. The first command performs a *clean* of previously built objects to avoid problems.

```
>del *.a *.o *.exe
>g++ -c adder.cpp
>ar rcs libmaths.a adder.o
>dir
    10/08/2017  13:50    <DIR>          .
    10/08/2017  13:50    <DIR>          ..
    10/08/2017  12:32                66 adder.cpp
    10/08/2017  12:45               575 adder.h
    10/08/2017  13:50               630 adder.o
    10/08/2017  12:44               350 calc.cpp
    10/08/2017  13:50             3,172 calc.o
    10/08/2017  12:44               128 hello.cpp
    10/08/2017  13:50               776 libmaths.a
                   7 File(s)         18,824 bytes
                   2 Dir(s)  92,681,625,600 bytes free
```

- You can see that we now have libmaths.a. Note that libmaths.a is slightly larger than the object from which it was created as it also contains an index describing the contents of the archive.
- Now create the object for calc.cpp and link it with the library we have created.

```
>g++ -c calc.cpp
>g++ calc.o libmaths.a -o calc.exe
>calc 6 7
    13
```

Libraries are usually stored in a common *libraries* or *lib* directory somewhere on the file system (look for a lib directory where you installed the gcc compiler, it will contain lots of these files), this means that you must tell g++ where to find the library. The header files associated with the library are usually stored in an *include* directory. Look at the following example, this is typical of the way `g++` would be used in a larger software project.

Normally libraries would be stored in a different location to the program that is being built. To demonstrate a more realitic project structure, first create a folder containing the library and copy *adder.h* and *libmaths.a* (the components of our maths library that the end user would need) to it.

```
>mkdir maths_library

>mkdir maths_library\lib
>move libmaths.a maths_libary\lib
    1 file(s) moved.

>mkdir maths_library\include
>copy adder.h maths_library\include
    1 file(s) moved

>mkdir calc_program
>copy calc.cpp calc_program
    1 file(s) moved

>del *.a *.o *.exe
```

The directory structure now looks something like:

```
Worksheet3
    |- calc_program                 <- code for the program you are writing
        |- calc.c
    |- maths_library                <- The library (typically this would have been downloaded, or comp
        |- lib
            |- libmaths.a
        | - include
            |- adder.h
```

Finally use go into the folder you created for the calc program and use g++ in a more realistic way to build *calc.exe*.

```
>cd calc_program
>g++ -c calc.cpp -I..\maths_library\include
>g++ calc.o -L..\maths_library\lib -lmaths -o calc.exe
```

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 **I**ncluded 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. (I in -I is a upper case i)
- When **linking** our program, we must:
  - Tell the linker where to find our **L**ibraries, We do this using the ***-Lpath_to_lib_directory*** linker option (where *path_to_lib_directory* is the path to the directory containing the .a files). You can have more than one -L option if libraries are in multiple locations.
  - Tell the linker which libraries to include. This is now done use the ***-llibrary_name*** option. Note that we give the name of the library (*maths*), not the name of the library file *libmaths.a*. This is GNU/Linux convention which is where g++ originated. (l in -l is a lower case L)

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.

Dynamic Libraries (Non-windows users see notes at end)

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*).

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

Exercise 5

- First we will modify our library header fole (adder.h) to make it compatible with a dynamic library. The reasons for these modifications are given below. Firstly, `cd` back the Week2 working directory and edit adder.h with the following modifications.

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

- Now build the dynamic library.

```
>g++ -c adder.cpp -Dmaths_EXPORTS
>g++ -shared adder.o -o libmaths.dll -Wl,--out-implib,libmaths.dll.a
>dir
    10/08/2017  15:58    <DIR>          .
    10/08/2017  15:58    <DIR>          ..
    10/08/2017  15:51                66 adder.cpp
    10/08/2017  15:39             1,364 adder.h
    10/08/2017  15:51               726 adder.o
    10/08/2017  12:44               350 calc.cpp
    10/08/2017  14:19            72,667 calc.exe
    10/08/2017  14:18             3,172 calc.o
    10/08/2017  12:44               128 hello.cpp
    10/08/2017  15:58             5,522 libmaths.dll
    10/08/2017  15:58             1,422 libmaths.dll.a
    10/08/2017  14:17    <DIR>          maths_library
    10/08/2017  14:19    <DIR>          calc_program
                9 File(s)         96,992 bytes
                4 Dir(s)  92,432,969,728 bytes free
```

- the `-Dmaths_EXPORTS` option defines the maths_EXPORTS variable for adder.h to use.
- the `-shared` option tells g++ to create a shared library and `-Wl,--out-implib,libmaths.dll.a` tells g++ to ask its linker to generate an *import library*.
- The two outputs from this process are:
  - **libmaths.dll** This is our dynamic/shared library and contains the adder code. It will be loaded/linked by calc_dynamic.exe at *run-time*.
  - **libmaths.dll.a** This is a small static import libary 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 *libmaths.dll* at runtime.
- Ok, so now update the library with the newly compiled files and header. It is standard practice for the *dll* files to reside in a *bin* directory as they are binary code.

```
>copy /Y adder.h maths_library\include
     1 file(s) moved.

>move libmaths.dll.a maths_library\lib
     1 file(s) moved.

>mkdir maths_library\bin
>move libmaths.dll maths_library\bin
         1 file(s) moved.
```

This should result in the directory structure below.

```
Worksheet3
    |- calc_program
        |- calc.c
    |- maths_library
        |- lib
            |- libmaths.a
            |- libmaths.dll.a
        |- include
            |- adder.h
        |- bin
            |-libmaths.dll
```

- Now create *calc_dynamic.exe* in exactly the same way as we created *calc.exe*, except that we will link against *libmaths.dll.a*, not *libmaths.a*. The `-Wl,Bdynamic` option does this, other than this option the command is the same as before.
- You can also force a static link with `-Wl,Bstatic`.

```
>cd calc_program
>g++ -c calc.cpp -I..\maths_library\include
>g++ calc.o -L..\maths_library\lib -lmaths -Wl,-Bdynamic -o calc_dynamic.exe
>dir
    10/08/2017  16:15    <DIR>          .
    10/08/2017  16:15    <DIR>          ..
    10/08/2017  15:51                66 adder.cpp
    10/08/2017  15:51               726 adder.o
    10/08/2017  12:44               350 calc.cpp
    10/08/2017  14:19            72,667 calc.exe
    10/08/2017  16:12             3,182 calc.o
    10/08/2017  16:15            72,759 calc_dynamic.exe
    10/08/2017  12:44               128 hello.cpp
    10/08/2017  16:07    <DIR>          maths_library
```

```
                         7 File(s)        161,453 bytes
                         3 Dir(s)  92,430,102,528 bytes free
```

- Now try to run *calc_dynamic.exe*

```
>calc_dynamic 6 7
```

- Does it work? If not, what is the problem?
- Fix it. (hint: what did you have to do for the MinGW *bin* directory at the beginning?)

The library that you have created has a typical library directory structure, and you will see something very similar if you look at the Qt library that you installed earlier (e.g. in `C:\Qt\5.9.1\mingw53_32`). You should now understand how a library such as Qt is organised and how it can be used in your code. This basic directory structure is central to unix-type operating systems, including Linux and MacOS.

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.

Analysing Dependencies

On Windows you can download a program called Dependency Walker 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.exe* and *calc_dynamic.exe*.

Dynamic Libraries (Other OSs)

On Linux dynamic libraries are handled a bit differently to on Windows.

Firstly they are usually called "Shared Objects" and have the .so extension (e.g. libmaths.so) but the main difference is how these operating systems deal with different versions of the same dynamic library. Windows struggles if there is more than 1 version of a dll since by convention, both versions will have the same name. E.g. mylibrary version 10 (*mylib.dll*) exists in `c:\mylib10\bin` and my library version 11 (also `mylib.dll`) exists in `c:\mylib11\bin`. Its entirely possible that my_program.exe, which was compiled to work with my library version 10, could find version 11 by mistake and fail to run properly. You can get around this by copying the correct version of `mylib.dll` to the same directory as `myprogram.exe`, but this kind of defeats the point of having shared libraries!

Linux deals with this by allowing multiple versions of a shared object to exists (with names such as `libmylib.so.11` and `libmylib.so.10`), there may also be a shortcut (known as a *symbolic link*) that points to the most recent version (`libmylib.so` is a shortcut to `libmylib.so.11`). Dynamic libraries are not located in the *bin* directory with Linux, they are located in the *lib* directory. Typically dynamic libraries that are a core operating system component will be in the */lib/* location, and dynamic libraries that are part of installed software will be in the */usr/lib/* location. A program called ldconfig tracks the location of installed shared libraries, and a program called ldd can be used to analyse which shared libraries are needed by a program.

You can also read more about this here.

Creating and using dynamic libraries on OSX (also a Unix derivative and uses similar file locations to above) is described in detail here.

Exercise 6

The compile and link process is fundamentally the same regardless of which compiler is used. You will now repeat exercise 4 and 5 but using Microsoft's compiler which are bundled with Visual Studio. Some hints on the differences between g++ and Microsoft's equivalent are given below: (If you are using a non-windows OS, you don't need to complete this exercise - just make sure you are comfortable using the compiler on your system to build executables, static and dynamic libraries and that you understand how your operating system deals with dynamic libraries.)

- Download Visual Studio (if it isnt already installed on your computer). You can obtain a free copy through Microsoft Imagine. You must now:
- Create a new directory (e,g. `Worksheet3\calc_msvc`) and copy adder.cpp, adder.h and calc.cpp into it.
- Build a static maths library using the Microsoft *cl* compiler.
- Build a statically linked *calc.exe* program using the *cl* compiler and *link* linker.
- Build a dynamic maths library using the Microsoft *cl* compiler.
- Build a dynamically linked *calc.exe* program using the *cl* compiler and *link* linker.

Hints for using the Microsoft build tools

- You must first ensure that the compiler `cl.exe /c` (use instead of `g++ -c`) and linker `link.exe` (use instead of g++ -o) and archiver `lib.exe` (equivalent to ar rcs) are on the path. Microsoft provides different versions of these tools depending on the version of Visual Studio you are using and whether you want to build 32 or 64 bit programs. To simplify setting the path for the specific version of cl and link that you want to use, it provides some batch scripts that set the path for you.
  - For older Visual Studio versions (<2017), the location of the batch file will be something like: `C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat`
  - For Visual Studio 2017, it will be something like: `C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\vcvarsall.bat`
  - More detail about the scripts can be found on msdn. Check you know how to use the scripts to setup both 32 and 64 bit builds.
- Step 1 after you open a cmd terminal is to run the vcvarsall script to setup build environment.

```
>"C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat" x86
```

- For Microsoft tools, options are usually specified using a `/` rather than a '-'.
- For `cl`: -c becomes /c, the /Fo*OBJECT_NAME* option can be used to specify object names, you should also specify the /EHsc option to avoid warnings. /Dmaths_STATIC and /Dmaths_EXPORTS options should be used when compiling statically linked executables or static libraries, or for a DLL, respectively. The dllimport/dllexport keywords are critical when using dynamic libraries in MSVC++.
- For `link`: the /out:*OUTPUT_NAME* option can be used to specify the output executable or dll name, and the /dll option is used to create dynamic libraries and import libraries.
- For `lib`: the /out:*STATIC_LIB_NAME* option can be used to specify the output library name.
- Usually, static libraries are named *.lib, dynamic libraries *.dll, objects *.obj (you can override this, but its the convention with Microsoft compilers)
- If you run into problems, Google it!