H62PEP Computing - Worksheet 4 - Build Systems and CMake

P. Evans

Automating the Build - Build Systems

For a program with many source files and many library dependencies, a large number of long commands must be sent to the compiler and linker. These need to be automated in some way. You could do this in a number of simple ways such are writing Batch Files or Shell Scripts. You could also use a **build system**.

A **build system** is a method for sending a predetermined sequence of instructions to a compiler and linker. Common build systems include *make* which is a utility that is included with g++ and can automate the build, it takes *Makefiles* as an input. IDEs such as Visual Studio, Eclipse, Code::Blocks, etc are also build systems, they take their own specific format of *Project* files as inputs.

The problem with all of these approaches is that they require you to know specific details of the platform that will be used to compiling the software *in advance*, details such as the operating system, compiler (remember the compiler and linker commands are different for gcc and msvc), build system (Makefiles or IDE, which IDE?), and the path to all required libraries (headers and static libraries). This is rarely possible and so another solution is needed. Furthermore, for true cross-platform support you would need to generate build process information for every possible combination.

Cross-Platform Support - Build System Generators

Build system generators introduce an additional step into the build process. You no longer write instructions for the compiler and linker, or project files for the build system, you write a generic recipe for a build system generator. You bundle your recipe with the source code and anyone who wants to build your software project first runs the build system generator which converts the recipe to a bespoke set of build instructions for their chosen build system. Here's the process in a bit more detail:

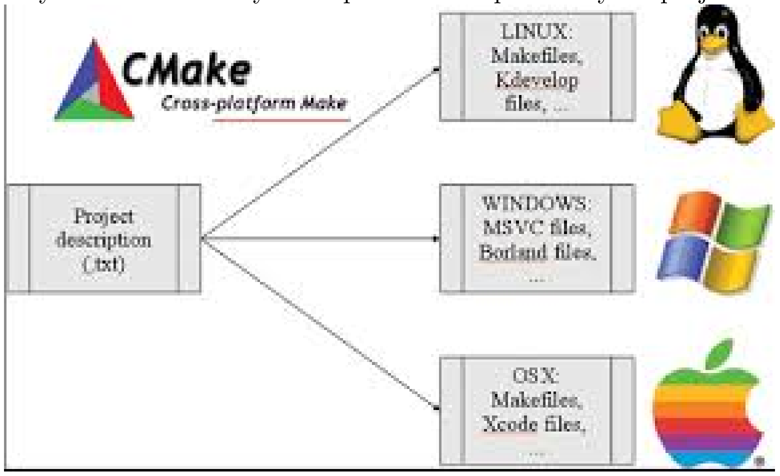
- **Build Recipe** You write a generic recipe for a build system generator. It lists the outputs required (executables, static libraries, dynamic libraries etc), the source files needed to build these outputs, and the names and minimum version numbers of libraries required to link the project. It may also contain information about how to build documentation for your project, how to install your project, and many other things (we'll cover these things later in the year).
- **The *configure* process** Another user obtains your source code and build recipe. They run the build system generator which reads the recipe and analyses their system. It determines which operating system the user is compiling for, which compiler they want to use, which build system they will use, checks to see if they have the necessary libraries installed, and determines the location of all header files (for `g++ -I`) and library files (for `g++ -L -l`). It might ask the user to specify some options for the build such as if they want static or dynamic linking. Finally it generates input for the users build system. For g++/make these instructions will be a set of Makefiles, for MS Visual Studio it could be a set of Makefiles or even a fully configured Visual Studio Project.
- **The *build* or *make* process** The build instructions or project are passed to the build environment which in turn executes the compiler and linker. In Visual Studio this could be by clicking build within the new project or by running `msbuild`, if Makefiles have been generated, `make` or `nmake` is used to interpret the Makefile instructions and execute the build.

A number of build systems generators exist which you should be aware of, including: * CMake - A fairly modern build system with its own GUI. It works nicely on a variety of operating systems and with most compilers including Visual Studio. Can generate installers for a range of operating systems. * Autotools - The GNU/Linux build system generator and the default for Linux software distributed as source. Not really cross-platform, it only supports GCC but does this across multiple OS and architectures. Not so easy to use with Windows (especially if you also want to use an IDE) but can be used through MSys. * Meson Newer build system generator. Supposed to be lightweight and fast. Claims good cross-platform support like CMake. Starting to replace Autotools for some GNU applications. * And many more...

CMake - A Simple Example

For this project we will use CMake. Do not confuse cmake with make!: CMake is a build system generator, make is a build system.

Many modern projects use CMake but it requires familiarity with many written and unwritten conventions to maximise your milage. The input of cmake is a set of files called *CMakeLists.txt* (one for each project directory) that provide a system and build system independent description of your project. The output of cmake is a system and build system specific description of your project. Quite a number of build systems are supported.


CMake
Cross-platform Make
Project description (.txt)
LINUX: Makefiles, Kdevelop files, ...
WINDOWS: MSVC files, Borland files, ...
OSX: Makefiles, Xcode files, ...

The following exercises show output for Windows but CMake behaves in almost the same way on all operating systems. If you're a non-windows person, you don't need run the final Visual Studio based exercises (Exercises 2,4) yourself if you don't want to but you should at least make sure you understand the differences in compiling software on different operating systems, and therefore why applications like CMake exist for cross-platform development.

Exercise 1

- Download and install CMake
- Make sure that the path to *cmake.exe* program is present in your system PATH environment variable.
- Create a new directory in your Individual repo called Worksheet4 and a subdirectory called *hello_cmake* and finally make a copy of *hello.cpp* in this directory.

```
>mkdir hello_cmake
>copy hello.cpp hello_cmake
            1 file(s) copied.
>cd hello_cmake
```

- Now we need to create a simple CMakeLists.txt. This is our platform and build system independent build recipe.

```
# hello_cmake/CMakeLists.txt
cmake_minimum_required( VERSION 2.8 )
project( hello )
add_executable( hello hello.cpp )

# / hello_cmake/CMakeLists.txt
```

- Note that we have define the name of our program but there's no mention of .exe and there's no mention of compiler or linker program names as both of these things are platform or build system dependent.
- Now we need to invoke `cmake` on the project. To do this, `cmake` needs to know which build system it should generate input for. A wide variety of build systems and other options are supported and you see options by typing:

```
>cmake --help

    [ Output omitted ]
    Generators

    The following generators are available on this platform:
    Visual Studio 15 2017 [arch] = Generates Visual Studio 2017 project files.
                                   Optional [arch] can be "Win64" or "ARM".
    Visual Studio 14 2015 [arch] = Generates Visual Studio 2015 project files.
                                   Optional [arch] can be "Win64" or "ARM".
    Visual Studio 12 2013 [arch] = Generates Visual Studio 2013 project files.
                                   Optional [arch] can be "Win64" or "ARM".
    Visual Studio 11 2012 [arch] = Generates Visual Studio 2012 project files.
                                   Optional [arch] can be "Win64" or "ARM".
    Visual Studio 10 2010 [arch] = Generates Visual Studio 2010 project files.
                                   Optional [arch] can be "Win64" or "IA64".
    Visual Studio 9 2008 [arch]  = Generates Visual Studio 2008 project files.
                                   Optional [arch] can be "Win64" or "IA64".
    Visual Studio 8 2005 [arch]  = Deprecated.  Generates Visual Studio 2005
                                   project files.  Optional [arch] can be
                                   "Win64".
    Borland Makefiles            = Generates Borland makefiles.
    NMake Makefiles              = Generates NMake makefiles.
    NMake Makefiles JOM          = Generates JOM makefiles.
    Green Hills MULTI            = Generates Green Hills MULTI files
                                   (experimental, work-in-progress).
    MSYS Makefiles               = Generates MSYS makefiles.
    MinGW Makefiles              = Generates a make file for use with
                                   mingw32-make.
    Unix Makefiles               = Generates standard UNIX makefiles.
    Ninja                        = Generates build.ninja files.
    Watcom WMake                 = Generates Watcom WMake makefiles.
    CodeBlocks - MinGW Makefiles = Generates CodeBlocks project files.
    CodeBlocks - NMake Makefiles = Generates CodeBlocks project files.
    CodeBlocks - NMake Makefiles JOM
                                 = Generates CodeBlocks project files.
    CodeBlocks - Ninja           = Generates CodeBlocks project files.
    CodeBlocks - Unix Makefiles  = Generates CodeBlocks project files.
    CodeLite - MinGW Makefiles   = Generates CodeLite project files.
    CodeLite - NMake Makefiles   = Generates CodeLite project files.
    CodeLite - Ninja             = Generates CodeLite project files.
    CodeLite - Unix Makefiles    = Generates CodeLite project files.
    Sublime Text 2 - MinGW Makefiles
                                 = Generates Sublime Text 2 project files.
    Sublime Text 2 - NMake Makefiles
                                 = Generates Sublime Text 2 project files.
    Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files.
    Sublime Text 2 - Unix Makefiles
                                 = Generates Sublime Text 2 project files.
    Kate - MinGW Makefiles       = Generates Kate project files.
    Kate - NMake Makefiles       = Generates Kate project files.
    Kate - Ninja                 = Generates Kate project files.
    Kate - Unix Makefiles        = Generates Kate project files.
    Eclipse CDT4 - NMake Makefiles
                               = Generates Eclipse CDT 4.0 project files.
    Eclipse CDT4 - MinGW Makefiles
                               = Generates Eclipse CDT 4.0 project files.
    Eclipse CDT4 - Ninja         = Generates Eclipse CDT 4.0 project files.
    Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
```

- We'll use g++ as our compiler and linker to begin with. Our version of g++ is part of the MinGW package and the default build system for this in make so we'll ask CMake to generate Makefiles for MinGW.
- We'll also build in a separate build directory, this way its easy to keep the source code separate from artefacts produced during the build.

```
>mkdir build
>cd build
>cmake -G "MinGW Makefiles" ..
    -- The C compiler identification is GNU 7.1.0
    -- The CXX compiler identification is GNU 7.1.0
    -- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe
    -- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe -- works
    -- Detecting C compiler ABI info
    -- Detecting C compiler ABI info - done
    -- Detecting C compile features
    -- Detecting C compile features - done
    -- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe
    -- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe -- works
    -- Detecting CXX compiler ABI info
    -- Detecting CXX compiler ABI info - done
    -- Detecting CXX compile features
    -- Detecting CXX compile features - done
    -- Configuring done
    -- Generating done
    -- Build files have been written to: cmake_hello/build
```

- Notice that CMake has found the g++ compiler and performed a test to check it is functional.
- Now look to see what the process has generated:

```
> dir
    11/08/2017  10:03    <DIR>          .
    11/08/2017  10:03    <DIR>          ..
    11/08/2017  10:03            15,374 CMakeCache.txt
    11/08/2017  10:03    <DIR>          CMakeFiles
    11/08/2017  10:03             1,405 cmake_install.cmake
    11/08/2017  10:03             5,262 Makefile
                                   3 File(s)         22,041 bytes
                                   3 Dir(s)  90,157,780,992 bytes free
```

- The important file here is *Makefile* which contains the instructions for our compiler. At this stage the *configure* part of the build is done. CMake has done its job and generated instructions for our build system. We now need to do the build and in order to this we need to invoke the make program within MinGW. On Windows, make is actually called mingw32-make in most cases.

```
>mingw32-make
    Scanning dependencies of target hello
    [ 50%] Building CXX object CMakeFiles/hello.dir/hello.cpp.obj
    [100%] Linking CXX executable hello.exe
    [100%] Built target hello
>dir
```

```
    11/08/2017  11:01    <DIR>          .
    11/08/2017  11:01    <DIR>          ..
    11/08/2017  10:03            15,374 CMakeCache.txt
    11/08/2017  11:01    <DIR>          CMakeFiles
    11/08/2017  10:03             1,405 cmake_install.cmake
    11/08/2017  11:01           392,263 hello.exe
    11/08/2017  10:03             5,262 Makefile
                   4 File(s)        414,304 bytes
                   3 Dir(s)  91,090,804,736 bytes free
```

- We now have hello.exe and you can see from the output of mingw32-make that the *build object* and *link output* stages have both been performed.
- By default, we can't see the commands that were actually executed so let's run make again but get it to show us what was going on.

```
>mingw32-make clean
>mingw32-make VERBOSE=1
    "C:\Program Files\CMake\bin\cmake.exe" -H"C:\Users\ezzpe\Dropbox (Personal)\Teac
    hing\H62PEP - Software\Week2\source\hello_cmake" -B"C:\Users\ezzpe\Dropbox (Pers
    onal)\Teaching\H62PEP - Software\Week2\source\hello_cmake\build" --check-build-s
    ystem CMakeFiles\Makefile.cmake 0
    "C:\Program Files\CMake\bin\cmake.exe" -E cmake_progress_start "C:\Users\ezzpe\D
    ropbox (Personal)\Teaching\H62PEP - Software\Week2\source\hello_cmake\build\CMak
    eFiles" "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\sour
    ce\hello_cmake\build\CMakeFiles\progress.marks"
    mingw32-make -f CMakeFiles\Makefile2 all
    mingw32-make[1]: Entering directory 'C:/Users/ezzpe/Dropbox (Personal)/Teaching/
    H62PEP - Software/Week2/source/hello_cmake/build'
    mingw32-make -f CMakeFiles\hello.dir\build.make CMakeFiles/hello.dir/depend
    mingw32-make[2]: Entering directory 'C:/Users/ezzpe/Dropbox (Personal)/Teaching/
    H62PEP - Software/Week2/source/hello_cmake/build'
    "C:\Program Files\CMake\bin\cmake.exe" -E cmake_depends "MinGW Makefiles" "C:\Us
    ers\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\source\hello_cmake
    " "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\source\hel
    lo_cmake" "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\so
    urce\hello_cmake\build" "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Sof
    tware\Week2\source\hello_cmake\build" "C:\Users\ezzpe\Dropbox (Personal)\Teachin
    g\H62PEP - Software\Week2\source\hello_cmake\build\CMakeFiles\hello.dir\DependIn
    fo.cmake" --color=
    mingw32-make[2]: Leaving directory 'C:/Users/ezzpe/Dropbox (Personal)/Teaching/H
    62PEP - Software/Week2/source/hello_cmake/build'
    mingw32-make -f CMakeFiles\hello.dir\build.make CMakeFiles/hello.dir/build
    mingw32-make[2]: Entering directory 'C:/Users/ezzpe/Dropbox (Personal)/Teaching/
    H62PEP - Software/Week2/source/hello_cmake/build'
    [ 50%] Building CXX object CMakeFiles/hello.dir/hello.cpp.obj
    C:\msys64\mingw64\bin\g++.exe     -o CMakeFiles\hello.dir\hello.cpp.obj -c "C:\U
    sers\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\source\hello_cmak
    e\hello.cpp"
    [100%] Linking CXX executable hello.exe
    "C:\Program Files\CMake\bin\cmake.exe" -E cmake_link_script CMakeFiles\hello.dir
    \link.txt --verbose=1
    "C:\Program Files\CMake\bin\cmake.exe" -E remove -f CMakeFiles\hello.dir/objects
    .a
    C:\msys64\mingw64\bin\ar.exe cr CMakeFiles\hello.dir/objects.a @CMakeFiles\hello
    .dir\objects1.rsp
    C:\msys64\mingw64\bin\g++.exe    -Wl,--whole-archive CMakeFiles\hello.dir/object
    s.a -Wl,--no-whole-archive  -o hello.exe -Wl,--out-implib,libhello.dll.a -Wl,--m
    ajor-image-version,0,--minor-image-version,0 @CMakeFiles\hello.dir\linklibs.rsp
    mingw32-make[2]: Leaving directory 'C:/Users/ezzpe/Dropbox (Personal)/Teaching/H
    62PEP - Software/Week2/source/hello_cmake/build'
    [100%] Built target hello
    mingw32-make[1]: Leaving directory 'C:/Users/ezzpe/Dropbox (Personal)/Teaching/H
    62PEP - Software/Week2/source/hello_cmake/build'
    "C:\Program Files\CMake\bin\cmake.exe" -E cmake_progress_start "C:\Users\ezzpe\D
    ropbox (Personal)\Teaching\H62PEP - Software\Week2\source\hello_cmake\build\CMak
    eFiles" 0
```

- The first commands that were executed as part of the build were to cmake to do some final checking and ensure the Makefiles are up-to-date.
- If you look at the later commands, you should see some familiar commands being issued to `g++` and `ar`. These commands are more complex than the ones we used, but achieve the same result. CMake is following a standard pattern for the build process that it knows will allow large or small projects to build successfully with MinGW and `g++`.

Port to Visual Studio

Exercise 2

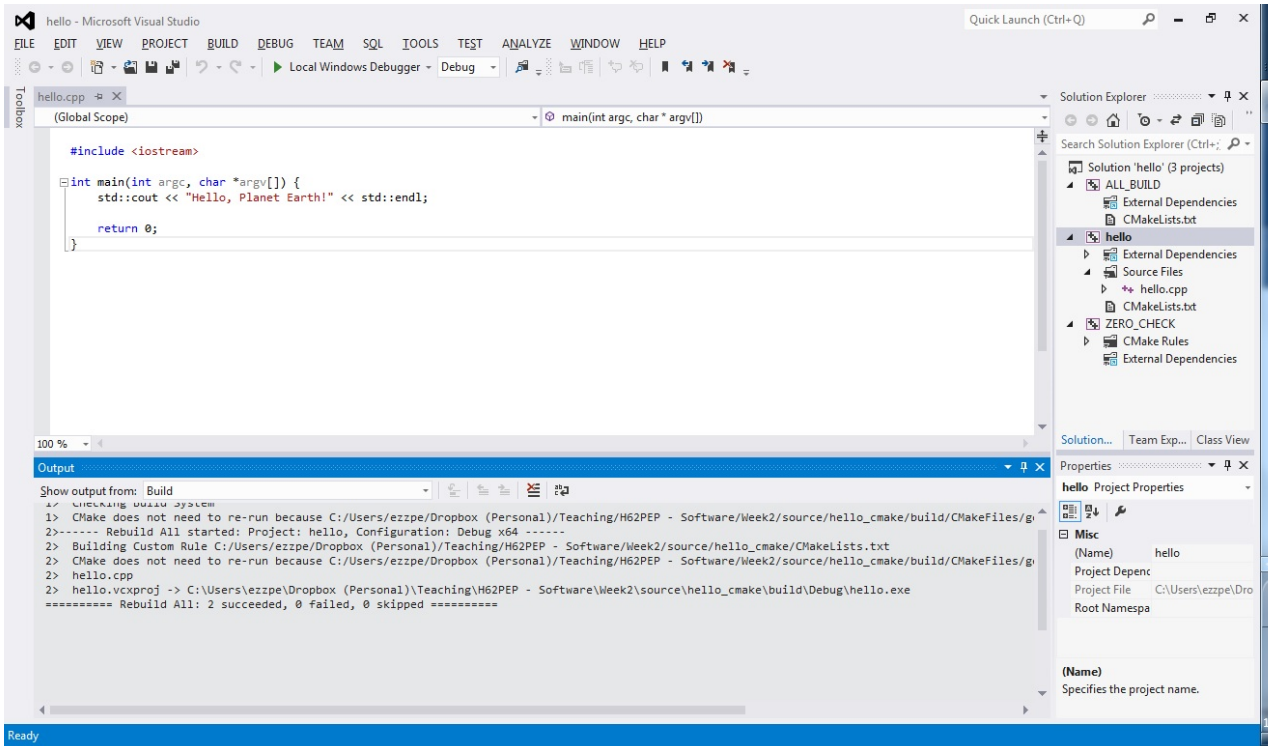
- Now ask CMake to generate input for Visual Studio and its compiler. Create a second build directory in the hello_cmake directory and rerun CMake in it but specifying Visual Studio as the compiler. You'll need to specify the version of Visual Studio you downloaded earlier.

```
>cmake -G "Visual Studio 11 2012 Win64" ..
    -- The C compiler identification is MSVC 17.0.50727.1
    -- The CXX compiler identification is MSVC 17.0.50727.1
    -- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio
    11.0/VC/bin/x86_amd64/cl.exe
    -- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio
    11.0/VC/bin/x86_amd64/cl.exe -- works
    -- Detecting C compiler ABI info
    -- Detecting C compiler ABI info - done
    -- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studi
    o 11.0/VC/bin/x86_amd64/cl.exe
    -- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studi
    o 11.0/VC/bin/x86_amd64/cl.exe -- works
    -- Detecting CXX compiler ABI info
    -- Detecting CXX compiler ABI info - done
    -- Detecting CXX compile features
    -- Detecting CXX compile features - done
    -- Configuring done
    -- Generating done
    -- Build files have been written to: hello_cmake/build
>dir
        11/08/2017  11:39    <DIR>          .
        11/08/2017  11:39    <DIR>          ..
        11/08/2017  11:39            59,237 ALL_BUILD.vcxproj
        11/08/2017  11:39               339 ALL_BUILD.vcxproj.filters
        11/08/2017  11:39            12,904 CMakeCache.txt
        11/08/2017  11:39    <DIR>          CMakeFiles
        11/08/2017  11:39             1,406 cmake_install.cmake
        11/08/2017  11:39             3,099 hello.sln
        11/08/2017  11:39            70,179 hello.vcxproj
        11/08/2017  11:39               683 hello.vcxproj.filters
        11/08/2017  11:39            40,903 ZERO_CHECK.vcxproj
        11/08/2017  11:39               582 ZERO_CHECK.vcxproj.filters
                                    9 File(s)        189,332 bytes
                                    3 Dir(s)  91,094,753,280 bytes free
```

- We now have all of the files needed to build our project using Visual C++ and/or Microsofts C++ compiler. First the correct build programs need adding to PATH.

```
>"C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat" amd64
>msbuild hello.sln
    Microsoft (R) Build Engine version 4.6.1087.0
    [Microsoft .NET Framework, version 4.0.30319.42000]
    Copyright (C) Microsoft Corporation. All rights reserved.

    Building the projects in this solution one at a time. To enable parallel build,
    please add the "/m" switch.
    Build started 11/08/2017 11:47:04.
    Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\sou
    rce\hello_cmake\build\hello.sln" on node 1 (default targets).
    ValidateSolutionConfiguration:
    Building solution configuration "Debug|X64".
    ValidateProjects:
    The project "ALL_BUILD" is not selected for building in solution configuratio
    n "Debug|x64".
    Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\sou
    rce\hello_cmake\build\hello.sln" (1) is building "C:\Users\ezzpe\Dropbox (Perso
    nal)\Teaching\H62PEP - Software\Week2\source\hello_cmake\build\ZERO_CHECK.vcxpr
    oj" (2) on node 1 (default targets).
    PrepareForBuild:
    Creating directory "x64\Debug\ZERO_CHECK\".
    InitializeBuildStatus:
    Creating "x64\Debug\ZERO_CHECK\ZERO_CHECK.unsuccessfulbuild" because "AlwaysC
    reate" was specified.
    CustomBuild:
    Checking Build System
    CMake does not need to re-run because C:/Users/ezzpe/Dropbox (Personal)/Teach
    ing/H62PEP - Software/Week2/source/hello_cmake/build/CMakeFiles/generate.stam
    p is up-to-date.
    FinalizeBuildStatus:
    Deleting file "x64\Debug\ZERO_CHECK\ZERO_CHECK.unsuccessfulbuild".
    Touching "x64\Debug\ZERO_CHECK\ZERO_CHECK.lastbuildstate".
    Done Building Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Soft
    ware\Week2\source\hello_cmake\build\ZERO_CHECK.vcxproj" (default targets).

    Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\sou
    rce\hello_cmake\build\hello.sln" (1) is building "C:\Users\ezzpe\Dropbox (Perso
    nal)\Teaching\H62PEP - Software\Week2\source\hello_cmake\build\hello.vcxproj.me
    taproj" (3) on node 1 (default targets).
    Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\sou
    rce\hello_cmake\build\hello.vcxproj.metaproj" (3) is building "C:\Users\ezzpe\D
    ropbox (Personal)\Teaching\H62PEP - Software\Week2\source\hello_cmake\build\hel
    lo.vcxproj" (4) on node 1 (default targets).
    PrepareForBuild:
    Creating directory "hello.dir\Debug\".
    Creating directory "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Softw
    are\Week2\source\hello_cmake\build\Debug\".
    InitializeBuildStatus:
    Creating "hello.dir\Debug\hello.unsuccessfulbuild" because "AlwaysCreate" was
    specified.
    CustomBuild:
    Building Custom Rule C:/Users/ezzpe/Dropbox (Personal)/Teaching/H62PEP - Soft
    ware/Week2/source/hello_cmake/CMakeLists.txt
    CMake does not need to re-run because C:/Users/ezzpe/Dropbox (Personal)/Teach
    ing/H62PEP - Software/Week2/source/hello_cmake/build/CMakeFiles/generate.stam
    p is up-to-date.
    ClCompile:
    C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\AMD64\CL.exe /c /Z
    i /nologo /W3 /WX- /Od /Ob0 /D WIN32 /D _WINDOWS /D "CMAKE_INTDIR=\"Debug\""
    /D _MBCS /Gm- /EHsc /RTC1 /MDd /GS /fp:precise /Zc:wchar_t /Zc:forScope /GR /
    Fo"hello.dir\Debug\\" /Fd"hello.dir\Debug\vc110.pdb" /Gd /TP /errorReport:que
    ue "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\source
    \hello_cmake\hello.cpp"
    hello.cpp
    Link:
    C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\AMD64\link.exe /ER
    RORREPORT:QUEUE /OUT:"C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Sof
    tware\Week2\source\hello_cmake\build\Debug\hello.exe" /INCREMENTAL /NOLOGO ke
    rnel32.lib user32.lib gdi32.lib winspool.lib shell32.lib ole32.lib oleaut32.l
    ib uuid.lib comdlg32.lib advapi32.lib /MANIFEST /MANIFESTUAC:"level='asInvoke
    r' uiAccess='false'" /manifest:embed /DEBUG /PDB:"C:/Users/ezzpe/Dropbox (Per
    sonal)/Teaching/H62PEP - Software/Week2/source/hello_cmake/build/Debug/hello.
    pdb" /SUBSYSTEM:CONSOLE /TLBID:1 /DYNAMICBASE /NXCOMPAT /IMPLIB:"C:/Users/ezz
    pe/Dropbox (Personal)/Teaching/H62PEP - Software/Week2/source/hello_cmake/bui
    ld/Debug/hello.lib" /MACHINE:X64  /machine:x64 hello.dir\Debug\hello.obj
    hello.vcxproj -> C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software
    \Week2\source\hello_cmake\build\Debug\hello.exe
    FinalizeBuildStatus:
    Deleting file "hello.dir\Debug\hello.unsuccessfulbuild".
    Touching "hello.dir\Debug\hello.lastbuildstate".
    Done Building Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Soft
    ware\Week2\source\hello_cmake\build\hello.vcxproj" (default targets).

    Done Building Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Soft
    ware\Week2\source\hello_cmake\build\hello.vcxproj.metaproj" (default targets).

    Done Building Project "C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Soft
    ware\Week2\source\hello_cmake\build\hello.sln" (default targets).

    Build succeeded.
    0 Warning(s)
    0 Error(s)

    Time Elapsed 00:00:02.19
>debug\hello.exe
    Hello, Planet Earth!
```

- You can also use the Visual C++ GUI to build the project if you want. Open Visual Studio and select Open Project then navigate to your build directory. Open hello.vcxproj and use the menus to build.



```
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout << "Hello, Planet Earth!" << std::endl;

    return 0;
}
```

```
1>  Checking Build System
1>  CMake does not need to re-run because C:/Users/ezzpe/Dropbox (Personal)/Teaching/H62PEP - Software/Week2/source/hello_cmake/build/CMakeFiles/g
2>------ Rebuild All started: Project: hello, Configuration: Debug x64 ------
2>  Building Custom Rule C:/Users/ezzpe/Dropbox (Personal)/Teaching/H62PEP - Software/Week2/source/hello_cmake/CMakeLists.txt
2>  CMake does not need to re-run because C:/Users/ezzpe/Dropbox (Personal)/Teaching/H62PEP - Software/Week2/source/hello_cmake/build/CMakeFiles/g
2>  hello.cpp
2>  hello.vcxproj -> C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\source\hello_cmake\build\Debug\hello.exe
========== Rebuild All: 2 succeeded, 0 failed, 0 skipped ==========
```

CMake - Multiple Project Outputs

- Now we will recreate the calc and adder example from the previous worksheet using CMake. There are now two project outputs, the adder library and the calc executable.

Exercise 3

- First create a new project directory and within this create a copy of *adder.cpp*, *adder.h* and *calc.cpp*.
- Then create a new CMakeLists.txt for the project. The contents of each file is given below.

```
>cd ../..
>mkdir calc_cmake
```

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

#include "adder.h"

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_cmake/calc.cpp ---------------------------------

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

// calc_cmake/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 and 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. CMake
// does this for us.
#if defined(WIN32)|defined(_WIN32)
    #ifdef maths_STATIC
        #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 our function
MATHSLIB_API int add( int a, int b );
#endif
// / calc_cmake/adder.h ---------------------------------------
```

```
# calc_cmake/CMakeLists.txt
cmake_minimum_required( VERSION 2.8 )
project( calc )

# This project contains a library
add_library( maths adder.cpp )
# Note: you could force CMake to always build a static/shared library:
#add_library( maths STATIC adder.cpp )
#add_library( maths SHARED adder.cpp )

if( NOT BUILD_SHARED_LIBS )
    # if static libs are compiled we need
    # to somehow communicate that to the
    # source code. The definition will be
    # used to determine how MATHSLIB_API
        # is defined in adder.h
    add_definitions( -Dmaths_STATIC )
endif()

# It also contains an executable
add_executable( calc calc.cpp )

# The executable uses the library
target_link_libraries( calc maths )

# /calc_cmake/CMakeLists.txt
```

- Its worth noting that the maths_EXPORTS variable is defined automatically when the dynamic library is being built. It is used in *adder.h* to determine the correct type modifer to use for function definitions.
- We will perform 4 different builds each in its own build directory:
  - Static with MinGW
  - Dynamic with MinGW
  - Static with Visual Studio
  - Dynamic with Visual Studio

```
>cd calc_cmake
>mkdir mingw_static
>mkdir mingw_dynamic
>mkdir msvc_static
>mkdir msvc_dynamic

>cd mingw_static
>cmake -G "MinGW Makefiles" ..
    -- The C compiler identification is GNU 7.1.0
    -- The CXX compiler identification is GNU 7.1.0
    -- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe
    -- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe -- works
    -- Detecting C compiler ABI info
    -- Detecting C compiler ABI info - done
    -- Detecting C compile features
    -- Detecting C compile features - done
    -- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe
    -- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe -- works
    -- Detecting CXX compiler ABI info
    -- Detecting CXX compiler ABI info - done
    -- Detecting CXX compile features
    -- Detecting CXX compile features - done
    -- Configuring done
    -- Generating done
    -- Build files have been written to: calc_cmake/mingw_static
    >mingw32-make
    Scanning dependencies of target maths
    [ 25%] Building CXX object CMakeFiles/maths.dir/adder.cpp.obj
    [ 50%] Linking CXX static library libmaths.a
    [ 50%] Built target maths
    Scanning dependencies of target calc
    [ 75%] Building CXX object CMakeFiles/calc.dir/calc.cpp.obj
    [100%] Linking CXX executable calc.exe
    [100%] Built target calc

>cd ../mingw_dynamic
>cmake -G "MinGW Makefiles" -DBUILD_SHARED_LIBS=ON ..
    -- The C compiler identification is GNU 7.1.0
    -- The CXX compiler identification is GNU 7.1.0
    -- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe
    -- Check for working C compiler: C:/msys64/mingw64/bin/gcc.exe -- works
    -- Detecting C compiler ABI info
    -- Detecting C compiler ABI info - done
    -- Detecting C compile features
    -- Detecting C compile features - done
    -- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe
    -- Check for working CXX compiler: C:/msys64/mingw64/bin/g++.exe -- works
    -- Detecting CXX compiler ABI info
    -- Detecting CXX compiler ABI info - done
    -- Detecting CXX compile features
    -- Detecting CXX compile features - done
    -- Configuring done
    -- Generating done
    -- Build files have been written to: C:/Users/ezzpe/Dropbox (Personal)/Teaching/
    H62PEP - Software/Week2/source/calc_cmake/mingw_dynamic

    C:\Users\ezzpe\Dropbox (Personal)\Teaching\H62PEP - Software\Week2\source\calc_c
    make\mingw_dynamic>mingw32-make
    Scanning dependencies of target maths
    [ 25%] Building CXX object CMakeFiles/maths.dir/adder.cpp.obj
    [ 50%] Linking CXX shared library libmaths.dll
    [ 50%] Built target maths
    Scanning dependencies of target calc
    [ 75%] Building CXX object CMakeFiles/calc.dir/calc.cpp.obj
    [100%] Linking CXX executable calc.exe
    [100%] Built target calc
```

Exercise 4

- Now repeat the build yourself for Visual Studio in the relevant build directories and check that all is ok with all four examples.