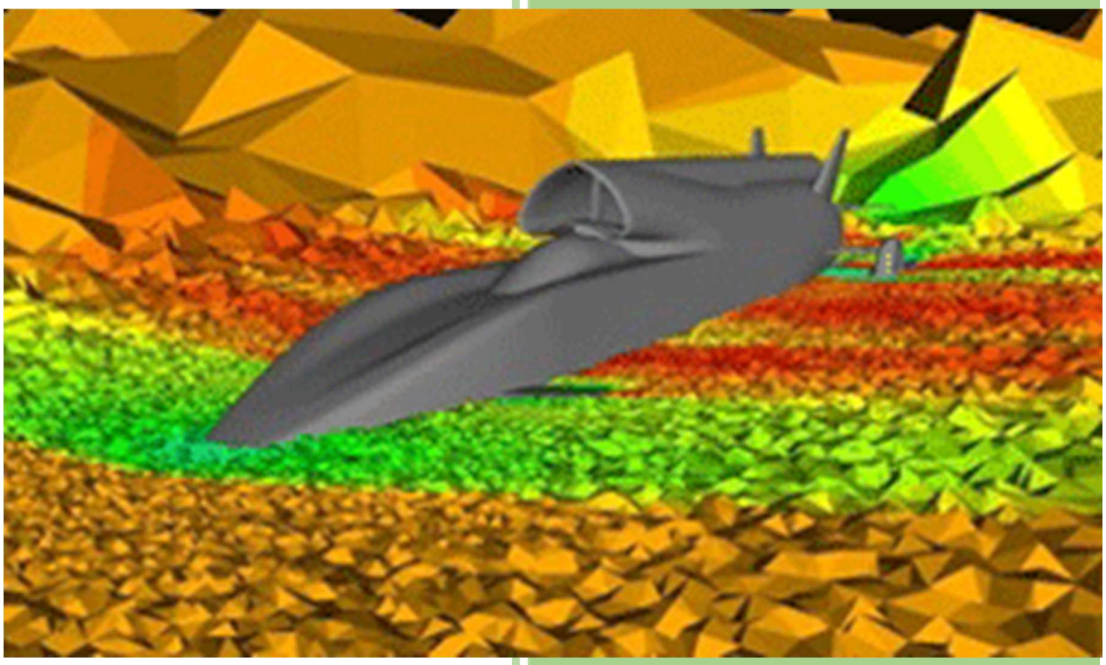
SwanSim CMake User Guide



Dr Jason W. Jones

College of Engineering, Swansea University

May 2017

Contents

1 Introduction

This document is part of a series of short guides available on the SwanSim web site (http://www.swansim.org).

2 Obtaining the Software

Obtaining CMake is a straightforward process regardless of the platform you are working on. It is available for download from https://cmake.org/download/ .

For Windows, they provide a simple installer which operates in the same way as installing any other Windows software. During the installation process, all of the options can be left as their defaults.

In Linux, it is often either pre-installed or available in the repository for each of the Linux distributions. Often, though, the version that is available can be out of date. This is particularly true for the repositories designed for servers such as RedHat and CentOS. If the version available is older than v3.0 then it is advisable to install a local copy from the website above. This can easily be achieved by installing one of the Linux binary distributions of CMake.

3 The Basics

CMake operates by reading a set of one or more files (CmakeLists.txt files) which describe how each component of the project is to be built from its constituent parts. For example, in its simplest form, an executable is built from one or more source code files and header files. Depending on the size and complexity of the project, this can be easily extended to build multiple libraries and then build executables which utilise those libraries.

In general, each component that is built is contained in a separate folder, each with a CMakeLists.txt file that describes how that component is built. For example, one folder would contain the source code for an executable, whereas separate folders would contain the source code for each library that is to be built.

The folder structure and the format of the CMakeLists.txt files will be familiar to anyone with experience of the classic Makefile.

CMake encourages *out-of-source* compilation. This means that the object files, libraries and executables that are created, as a result of the build process, are stored in a separate folder to that of the source code. This ensures that the folders containing the source code stay clean and are not polluted with transient files from the compilation. This also allows multiple versions of the software to easily be built in parallel.

To physically build your software, CMake is used to interpret these CMakeLists files and produce the files for your chosen combination of Operating System, IDE and compilers. CMake supports a large, and ever increasing, set of build tools including the production of Makefiles (Linux), project files for Visual Studio (Windows) and project files for XCode (Mac). As well as the production of Makefiles, CMake can also produce the project files for a number of common IDE's including, but not limited to, Code::Blocks, KDevelop3 and Eclipse.

4 Learn by Example

The quickest way to learn how to use a system is usually by example. This section contains several examples of using CMake to build a variety of programs. Each program is short and simple – just filling an array with numbers, doubling them and then printing them out.

4.1 A Simple Executable

The simplest example is one where a single executable is built from one or more source code files in one language.

4.1.1 C

The program for this example is shown below:

/double_array.c

```
#include "double_array.h"

int main( int argc, char *argv[] )
{
  float  numbers[10];
  int    i;

  printf( "We are about to double 10 numbers\n" );
  for( i = 0; i < 10; i++ ) {
    numbers[i] = (float)i * 25;
  }

  for( i = 0; i < 10; i++ ) {
    numbers[i] = numbers[i] * 2.0f;
  }

  for( i = 0; i < 10; i++ ) {
    printf( "Doubled Number is %f\n", numbers[i] );
  }
  return( 0 );
}
```

/double_array.h

```
#ifndef  __DOUBLE_ARRAY_H
#define  __DOUBLE_ARRAY_H

#include <stdio.h>

#endif  /*__DOUBLE_ARRAY_H*/
```

To build this using CMake, you need a 'CMakeLists.txt' file in the same folder. An example CMakeLists.txt file is shown below:

/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleArray )

set( C_SRC_FILES
  "double_array.c"
)

set( HEADER_FILES
  "double_array.h"
)

add_executable( DoubleArray
  ${C_SRC_FILES}
  ${HEADER_FILES}
)
```

The meaning of each of the lines is listed below:

Line 1 – This defines the minimum version of CMake that can build this project. Unless there is reason to do otherwise, choose version 3.0.

Line 3 – This defines the name of the Project. This is completely arbitrary and, except for the top-level CMakeLists file, optional.

Lines 5-7 – This defines a variable 'C_SRC_FILES' as a list of filenames (in this case just one filename). This is used later in the CMakeLists file to provide the list of C source code files. The name of the variable is entirely up to the user.

Lines 9-11 – This defines a variable 'HEADER_FILES' as a list of filenames. This is also used later in the CMakeLists file to provide the list of header files. The name of the variable is entirely up to the user.

Lines 13-16 – This tells CMake to build an executable called 'DoubleArray' using the files in 'C_SRC_FILES' AND 'HEADER_FILES'

4.1.2 Fortran

The same program as above but written in Fortran is shown below:

/double_array.f

```
      program DoubleNumbers

      real*4   numbers( 10 )
      integer  i

      print *,'We are about to double 10 numbers'
      do i = 1, 10
        numbers(i) = i * 25
      enddo

      do i = 1, 10
        numbers(i) = numbers(i) * 2.0
      enddo

      do i = 1, 10
        print *,'Doubled Number is ', numbers(i)
      enddo

      end
```

The CMakeLists.txt file is very similar to the one above. The new/changed lines are highlighted in red and explained below.

/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleArray )

enable_language( Fortran )

set( F77_SRC_FILES
  "double_array.f"
)

add_executable( DoubleArray
  ${F77_SRC_FILES}
)
```

Line 5 – This tells CMake that this project will be using a Fortran compiler. By default, CMake assumes that a project is made up of C and C++ files.

Lines 7-9 – This is similar to setting the C_SRC_FILES variable in the example above except, this time, it is more appropriate to name it F77_SRC_FILES. There is also no HEADER_FILES variable as Fortran, typically, doesn't use header files.

4.2 An Executable and a Library (Single Language)

Once a program gets large enough, it is appropriate to split it up into one or more libraries that get linked into the final executable. In this section, we will take the original program from the previous section and move the loop that doubles the array into a separate function in a separate library. The convention is to place all of the files for each library into a separate sub-directory.

4.2.1 C

In C, the files (along with their locations) are shown below:

/double_array.c

```
#include "double_array.h"

int main( int argc, char *argv[] )
{
  float  numbers[10];
  int    i;

  printf( "We are about to double 10 numbers\n" );
  for( i = 0; i < 10; i++ ) {
    numbers[i] = (float)i * 25;
  }

  doubleArray( numbers, 10 );

  for( i = 0; i < 10; i++ ) {
    printf( "Doubled Number is %f\n", numbers[i] );
  }
  return( 0 );
}
```

/double_array.h

```
#ifndef  __DOUBLE_ARRAY_H
#define  __DOUBLE_ARRAY_H

#include <stdio.h>
#include <double_array_lib.h>

#endif  /*__DOUBLE_ARRAY_H*/
```

/DoubleLib/double_array_lib.c

```
#include "double_array_lib.h"

void doubleArray( float numbers[], int numNumbers )
{
  int  i;

  for( i = 0; i < numNumbers; i++ ) {
    numbers[i] = numbers[i] * 2.0f;
  }
}
```

/DoubleLib/double_array_lib.h

```
#ifndef  __DOUBLE_ARRAY_LIB_H
#define  __DOUBLE_ARRAY_LIB_H

void doubleArray( float numbers[], int numNumbers );

#endif  /*__DOUBLE_ARRAY_LIB_H*/
```

This time there are two CMakeLists.txt files – one in the parent folder and one in the DoubleLib folder. Lines that have not already been explained in previous examples are highlighted in red and explained below.

/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleArray )

set( C_SRC_FILES
  "double_array.c"
)

set( HEADER_FILES
  "double_array.h"
)

add_subdirectory( DoubleLib )

include_directories( DoubleLib )


add_executable( DoubleArray
  ${C_SRC_FILES}
  ${HEADER_FILES}
)

target_link_libraries( DoubleArray
  MyLib
)
```

Line 13 – This instructs CMake to look in the 'DoubleLib' folder for a sub-project. This folder must have a CMakeLists.txt file in it.

Line 15 – This adds the specified folder to the list of paths that the compiler will search for header files.

Lines 22-24 – This informs CMake that when building the executable 'DoubleArray' that it must link with the library 'MyLib' (this is defined in the CMakeLists file in the sub-folder, shown below).

/DoubleLib/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleLib )

set( C_SRC_FILES
  "double_array_lib.c"
)

set( HEADER_FILES
  "double_array_lib.h"
)

add_library( MyLib
  ${C_SRC_FILES}
  ${HEADER_FILES}
)
```

This CMakeLists file relates to the sub-project that is responsible for building the library, 'MyLib'.

Line 3 – This line is optional as it is not the top-level CMakeLists file for the overall project. However, to ensure that each sub-folder is self-contained, it is recommended to define a project name.

Lines 13-16 - This informs CMake that a library, called 'MyLib' is to be built using the specified files.

4.2.2 Fortran

The same example as above can be performed easily in Fortran:

/double_array.f

```
      program DoubleNumbers

      real*4   numbers( 10 )
      integer  i

      print *,'We are about to double 10 numbers'
      do i = 1, 10
        numbers(i) = i * 25
      enddo

      call doubleArray( numbers, 10 )

      do i = 1, 10
        print *,'Doubled Number is ', numbers(i)
      enddo

      end
```

/DoubleLib/double_array_lib.f

```
      subroutine doubleArray( numbers, numNumbers )
      integer numNumbers
      real*4  numbers( numNumbers )

      do i = 1, 10
        numbers(i) = numbers(i) * 2.0
      enddo
      return
      end
```

Like the previous example, there are now two CMakeLists.txt files – one in the parent folder and one in the 'DoubleLib' folder.

/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleArray )

enable_language( Fortran )

set( F77_SRC_FILES
  "double_array.f"
)

add_subdirectory( DoubleLib )

add_executable( DoubleArray
  ${F77_SRC_FILES}
)

target_link_libraries( DoubleArray
  MyLib
)
```

/DoubleLib/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleLib )

enable_language( Fortran )

set( F77_SRC_FILES
  "double_array_lib.f"
)

add_library( MyLib
  ${F77_SRC_FILES}
)
```

4.3 An Executable and a Library (Mixed Language)

Whenever C/C++ and Fortran are mixed then there is always a difficulty with the way Fortran mangles it's variable and subroutine names as the way this is done depends on the platform and the compiler. If only Fortran is used then this is hidden from the user. However, when a Fortran subroutine needs to be called by C then the manner in which this is done becomes important.

For example, given the Fortran subroutine 'doubleArray', this could be changed to 'doublearray', 'DOUBLEARRAY', 'doublearray_', '_doublearray' and many other combinations.

CMake has a built-in system to detect this in each case. It then generates a header file (.h) that hides these details from you.

The CMakeFile.txt files are shown below:

/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleArray )

enable_language( Fortran )

set( C_SRC_FILES
  "double_array.c"
)

set( HEADER_FILES
  "double_array.h"
)

include( FortranCInterface )

FortranCInterface_HEADER( ${CMAKE_CURRENT_BINARY_DIR}/fc.h
                          MACRO_NAMESPACE "FC_" )

include_directories( ${CMAKE_CURRENT_BINARY_DIR} )

include_directories( DoubleLib )

add_subdirectory( DoubleLib )

add_executable( DoubleArray
  ${CXX_SRC_FILES}
  ${HEADER_FILES}
)

target_link_libraries( DoubleArray
  MyLib
)
```

Line 15 – This instructs CMake to include a built-in module called 'FortranCInterface'. This module provides the tools to automatically determine the name mangling of the chosen Fortran compiler.

Lines 17-18 – This tells CMake to produce a header file ‘fc.h’ which contains macros to perform the mapping from subroutine name to the mangled version. The MACRO_NAMESPACE parameter is prepended to the macros that are defined. (The use of this file is explained below in the source code)

Since the creation of this file is part of the build process, and is likely to depend on the platform used for the compilation, it should not be created in the source code folder. **The source code folder should be kept clean of any files that are part of the build process and, therefore, kept out of any source code repositories.** The variable, CMAKE_CURRENT_BINARY_DIR, always contains the path to the folder in which object files, libraries and executables will be stored for that project. This means that the ‘fc.h’ file will be stored in that folder.

Line 20 – In order to ensure the compiler finds the ‘fc.h’ header file whilst compiling, it’s path must be added to the compiler’s search path for include files.

/DoubleLib/CMakeLists.txt

```
cmake_minimum_required( VERSION 3.0 )

project( DoubleLib )

enable_language( Fortran )

set( F77_SRC_FILES
  "double_array_lib.f"
)

set( HEADER_FILES
  "double_array_lib.h"
)

add_library( MyLib
  ${F77_SRC_FILES}
  ${HEADER_FILES}
)
```

The source code files are listed below:

/double_array.c

```
#include "double_array.h"

int main( int argc, char *argv[] )
{
  float  numbers[10];
  int    i, numNumbers = 10;

  printf( "We are about to double 10 numbers\n" );
  for( i = 0; i < 10; i++ ) {
    numbers[i] = (float)i * 25;
  }

  doubleArray( numbers, &numNumbers );

  for( i = 0; i < 10; i++ ) {
    printf( "Doubled Number is %f\n", numbers[i] );
  }
  return( 0 );
}
```

In the above file, the function 'doubleArray' is called using its non-mangled name.

/double_array.h

```
#ifndef  __DOUBLE_ARRAY_H
#define  __DOUBLE_ARRAY_H

#include <stdio.h>
#include <double_array_lib.h>

#endif  /*__DOUBLE_ARRAY_H*/
```

/DoubleLib/double_array_lib.f

```
      subroutine doubleArray( numbers, numNumbers )
      integer numNumbers
      real*4  numbers( numNumbers )

      do i = 1, 10
        numbers(i) = numbers(i) * 2.0
      enddo
      return
      end
```

/DoubleLib/double_array_lib.h

```
#ifndef  __DOUBLE_ARRAY_LIB_H
#define  __DOUBLE_ARRAY_LIB_H

#include <fc.h>

#define doubleArray  FC_GLOBAL(doublearray,DOUBLEARRAY)

void doubleArray( float numbers[], int *pNumNumbers );

#endif  /*__DOUBLE_ARRAY_LIB_H */
```

The use of the 'fc.h' file happens in this header file. The 'fc.h' file contains a macro that requires two parameters:

1. The function name using all lower-case
2. The function name using all upper-case

The result of the FC_GLOBAL macro is then the correct function name to use in C/C++ to call the corresponding Fortran subroutine. (Please note, it is called FC_GLOBAL because we defined the namespace to be 'FC_').

In order to hide the complexities of the FC_GLOBAL macro from the rest of the code, another macros was defined, 'doubleArray', which is defined as the result of the FC_GLOBAL macro. This way the details of the Fortran name mangling are completely hidden to the rest of the C code.

5 Building a CMake Project

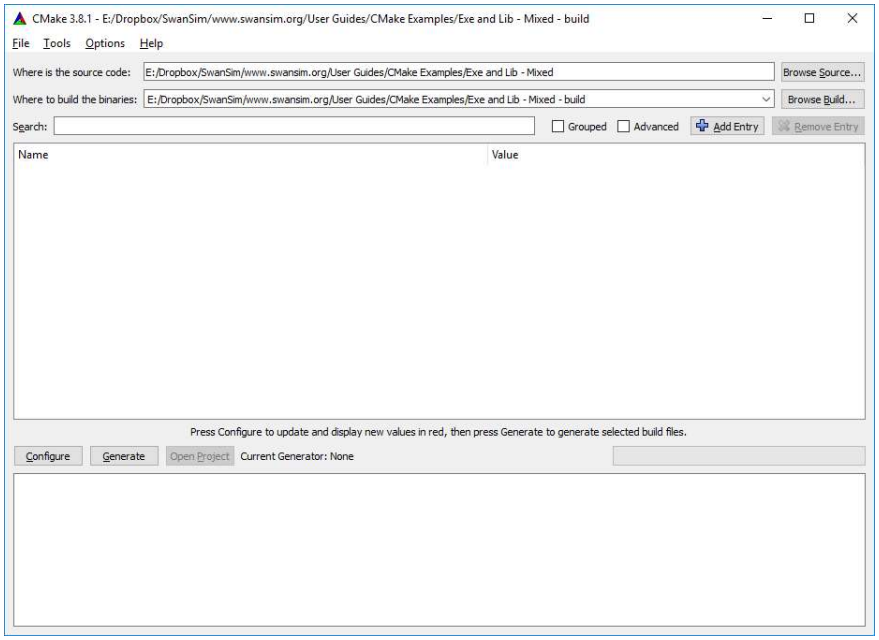
As explained previously, the purpose of CMake is to allow the build process of any project to be defined in a platform independent manner. In order to actually build the project, this must be converted into the native build files for the chosen platform and tool.

CMake supports a large, and ever increasing, combination of build tools. In this document, the four systems expected to be used the most are explained using the most complex of the examples above – the mixed language library and executable project.

5.1 Windows (Visual Studio Community 2015)

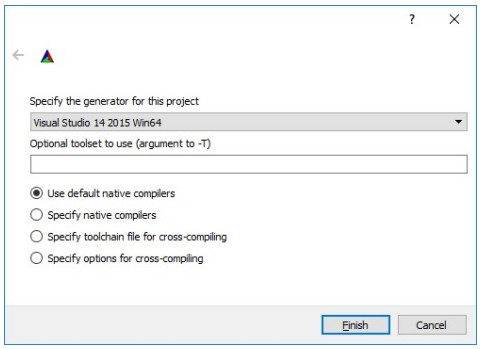
In Windows, the most popular software environment is Visual Studio. Microsoft provide a free version of this IDE with very few restrictions in its use. It should be noted, however, this does not include any Fortran compilers – these must be purchased separately. The most common compiler suite to use in conjunction with Visual Studio are the Intel compilers.

To use Visual Studio, the CMake-gui application must be executed. This opens a window in which you must enter the folder which contains the source code and the folder in which the results of the build will be stored as shown below:

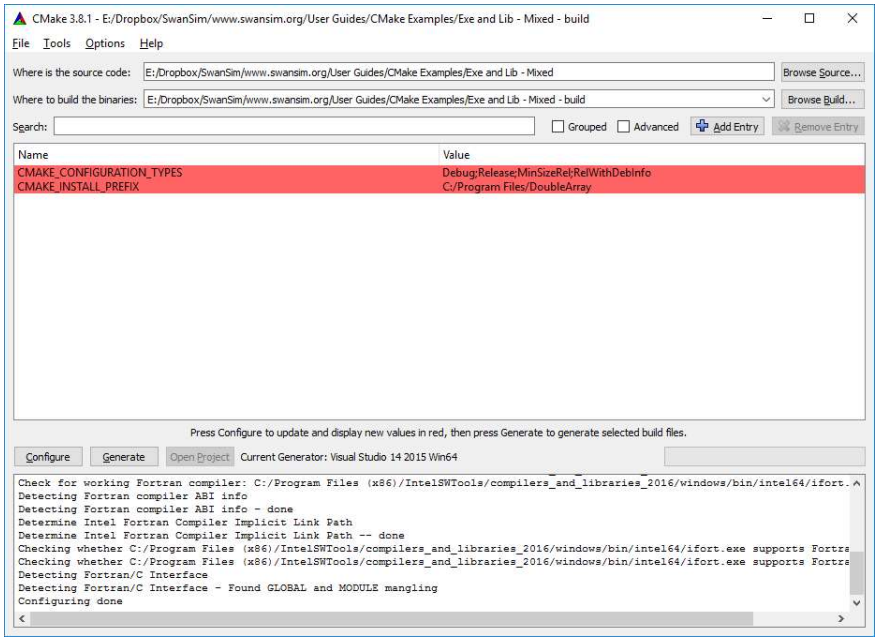


The next step is to select the 'Configure' button. This processes the CMakeLists files and then prompts you with any variables that it feels the user may want to edit. Any variables which are new are highlighted in red. For more complex projects, it may require more than one 'Configure' before all of the red variables are gone.

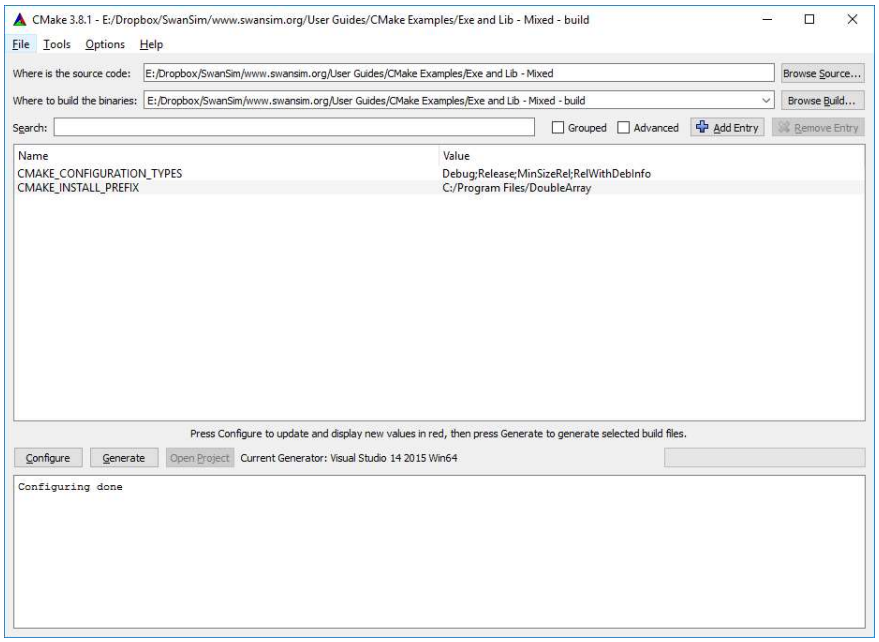
When 'Configure' is selected for the first time for a given project, it asks the user to select the Tools/IDE they wish to use. In this case, we will select 'Visual Studio 14 2015 Win64' as shown below:



Select ‘Finish’ and the configuration will continue after CMake has checked that it can find the selected compiler(s) and knows how to run them. This results in the window looking like this:

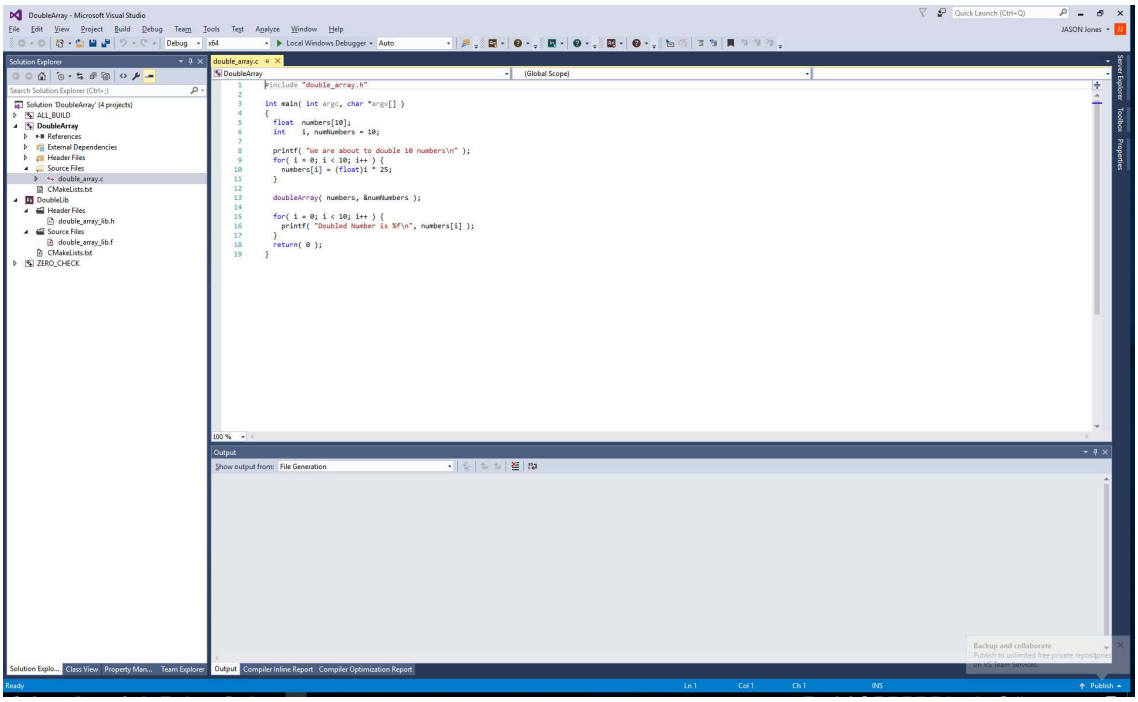


In this simple example, these variables can be left as their defaults so just click ‘Configure’ again.



Now, no variables are highlighted in red so we can generate the build environment by clicking on the 'Generate' button.

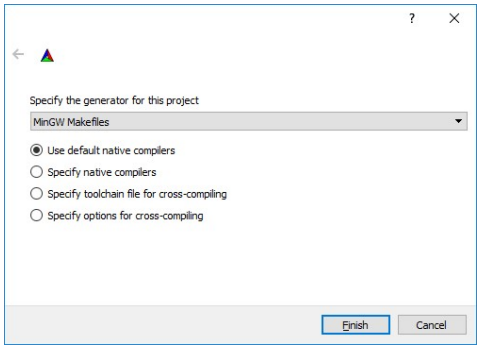
When this has finished, using a File Explorer, navigate into the folder you selected for the build and you will find a large set of files and folders have been created. These are the Visual Studio project files which can be opened by double-clicking on the Solution file (.sln).



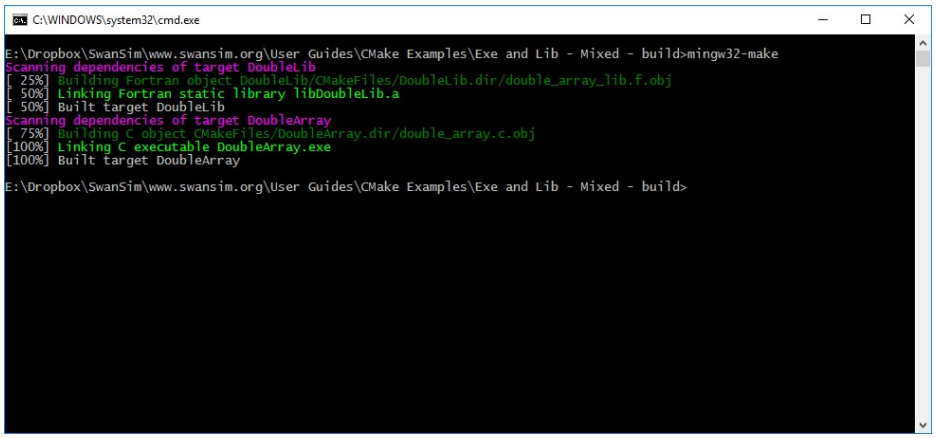
5.2 Windows (MinGW GNU Compilers)

This section assumes you have installed the MinGW-64 compilers as described in the 'Fortran Development in Windows' user guide.

The general CMake procedure is the same as for Visual Studio above. The only difference is the tools you select to build your project. In this case, 'MinGW Makefiles' should be chosen as shown below:

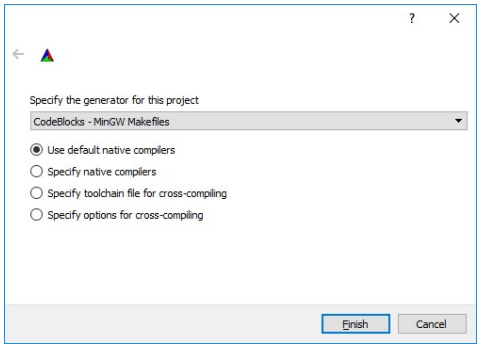


Once the normal 'Configure' and 'Generate' processes have been completed, use a Command Prompt to navigate to the build folder and enter the command 'mingw32-make'. The software will be built as shown below:

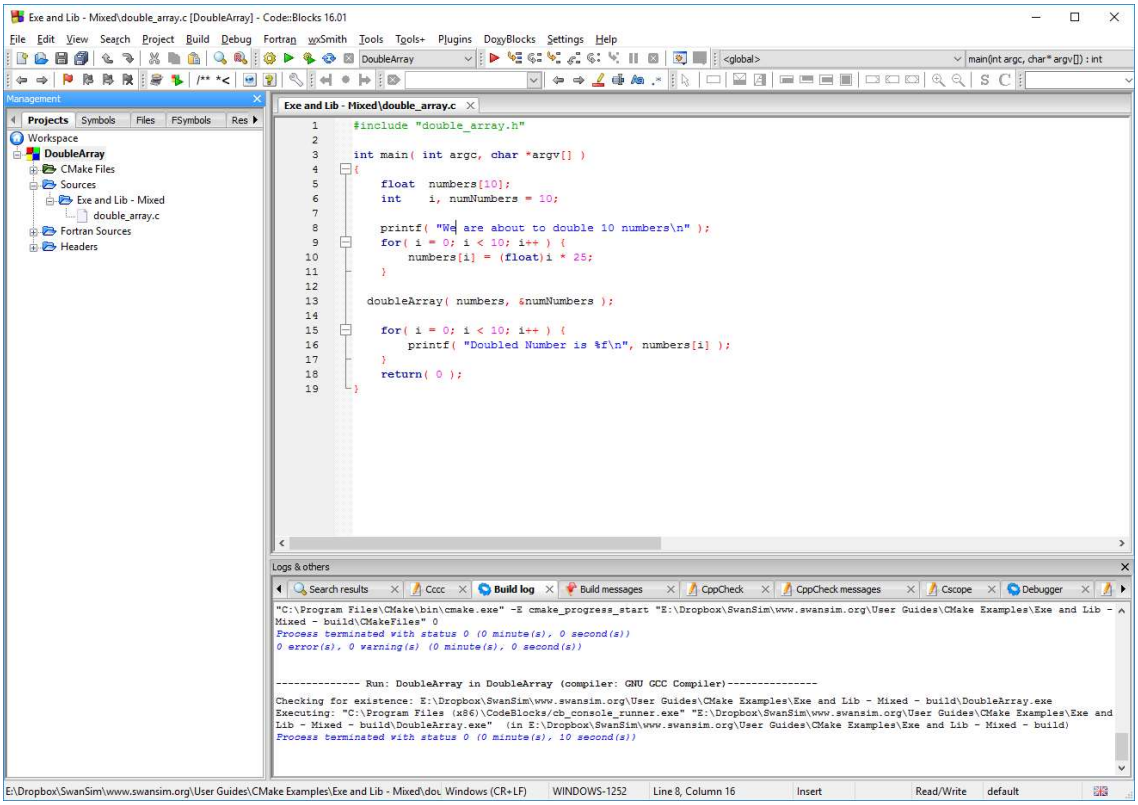


5.3 Windows (MinGW GNU Compilers and Code::Blocks IDE)

If the command-line method of building shown above is not desired, then the Code::Blocks IDE may be utilised instead. Ensure it is installed as described in the 'Fortran Development in Windows' user guide and then run CMake and select the option – 'CodeBlocks – MinGW Makefiles'.



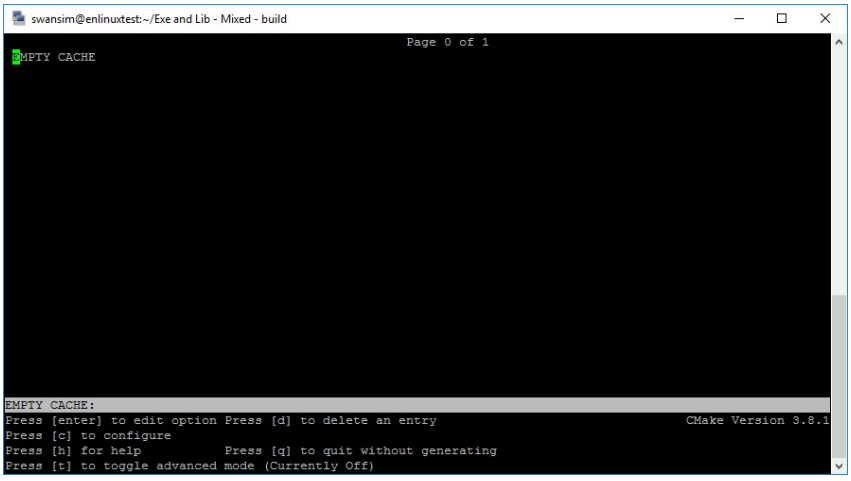
This causes the same makefiles to be produced but alongside that, Code::Blocks project files will be created. These can be loaded in Code::Blocks as shown below.



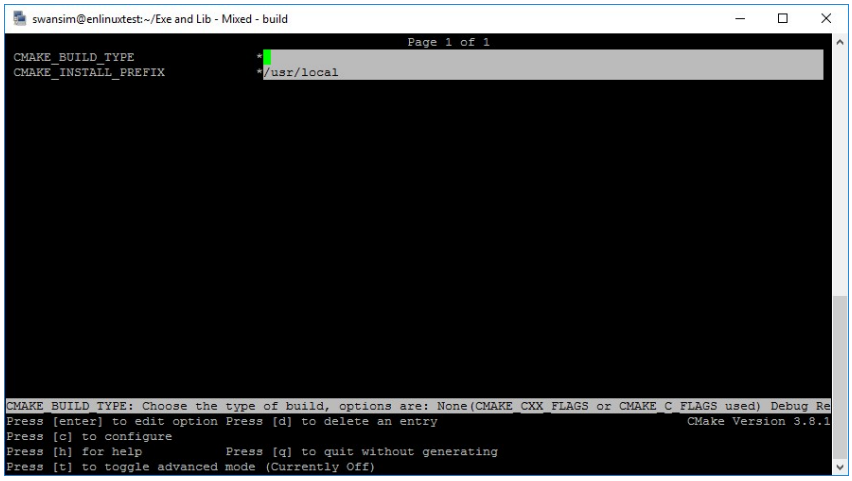
5.4 Linux (GNU Compilers and Makefiles)

The interface to CMake in Linux is less friendly and graphical than in Windows. The following steps must be followed:

1. Change to the build folder
2. Enter 'ccmake <path to the source folder>'
3. This will show a textual interface to CMake



4. Press 'c' to configure. This shows a set of variables with the values highlighted (inverted). This is showing they are new variables that need to be checked by the user (this is analogous to the red highlights in Windows). For this simple project these variables can be left as their defaults.



5. Press ‘c’ again and then press ‘g’ to generate the Makefiles. This exits ‘ccmake’
6. The project may now be built by using ‘make’.

6 Advanced: Finding and Linking External Libraries

6.1 OpenCascade

To be completed…..