Compiling and Building (including make)

Compiling and Building (1)

First step in the build process: compile the source code

The output from this step is generally known as the object code

Different compilers will produce different object codes from the same source code, and the naming conventions may be different.

Consequences:

- Use the same compiler for all source code
- Object files are .o or .obj

Compiling and Building (2)

Second step in the build process: link the object files

Except for the most simplest Fortran codes, most programs are built up from different pieces

The linker adds a number of extra files, the run-time libraries. Use -v if you want to see the gory details.

Stuff contained in the run-time libraries:

- input/output to the screen
- intrinsic functions (sin, cos, etc.)

Compiling and Building (3)

End result: an executable program!

Contains the compiled source code and various auxiliary routines that make it work

It also contains references to so-called dynamic run-time libraries (Windows: DLLs, Linux: shared objects or shared libaries)

Include files and Modules

Your program might be organized in some convenient directory tree. In this case, the compiler may need assistance in order to help it find everything.

- Fortran has the capability of including external files
- When compiling code that includes modules, the compiler will generate module intermediate files (.mod)

Compilers support the -I option to help locate these files.

Example: tabulate

Makefile Disclaimer

This course will give a brief overview of how to use make with Fortran

Will cover the basics only!

Motivation:

- You might be using an existing code that gets compiled with make
- You might want to incorporate this for your own projects/codes

What is Make?

Make is a tool which controls the generation of executables from a program's source files

It gets its knowledge of how to build your program from a file called the makefile

The compilation procedure is much faster!

- The compilation is done with a single command
- Only files that have been modified are recompiled
- Allows managing large programs with lots of dependencies

Makefile Basics (1)

A rule in the makefile tells Make how to execute a series of commands in order to build a target file from source files

It also specifies a list of dependencies of the target file

Here is what a simple rule looks like:

```
target : dependencies ... (also called prerequisites)
    <tab> commands
```

The <tab> is absolutely necessary!

Makefile Basics (2)

Make uses timestamps to locate the files that have been modified since the last time make was executed

By default when you type make it looks for the file makefile or Makefile. You can designate a specific name with make -f <thismakefile>

Can also use macros to give names to variables within the makefile. NOTE these are case-sensitive!

If no specific target is given in the make command then Make starts with the first target listed in the makefile

Let’s start with a very simple example (**example1**)

Makefile Basics (3)

Comments are delimited by the # symbol

A backslash \ can be used as a continuation character

Common extra tidbit: Create a "phony target" called clean which can be run to do a fresh recompile of all source code

Great reference: https://www.gnu.org/software/make/manual/make.html

Makefile Automatic Variables

These can only be values in the recipe. They cannot be used in the target list of a rule

$< The name of the first prerequisite

$^ The names of the all prerequisites

$@ The file name of the target of the rule

And there are even more available

Compiling Modules

When modules are compiled both a .o and .mod file are created

A .mod file is like a compiled header. This is what the compiler searches for when it sees a USE statement

The dependencies can start to get cumbersome and complicated when many modules are USED and inherited

Make has no method for determining these for you.

Take a look at **example2**

Compiling Modules (2)

If you edit a module but do not change the interface then there's no need to update the .mod file.

But this is compiler specific behavior:

gfortran has been updated to handle this

ifort always updates both the .o and .mod files

There are some software build tools that try to handle this complexities to try to reduce "cascading compilation".

Want it to compile fast, but really we want it correct!

Helpful Tools

mkDepends - generate a list of dependencies

mkSrcfiles - generate a list of all source files

Versions of these perl scripts are used in atmospheric models like SAM and CAM

mkdep - requires both GNU make and Python

fortran-lang.org has some excellent material on building programs including ways to generate dependency lists

C-preprocessing Blocks

A mechanism to include conditionally-compiled code

Will automatically be handled by using a file extension of .F, .FOR, .FTN, .fpp, .Fpp, .F90, .F95, .F03 or .F08

To manually invoke the preprocessor use -cpp

To activate the named blocks of code use -D<name>

Sample program: cpreproc.f90