Makefile learning tutorial for Fortran

This tutorial is about creating makefiles for the Fortran 90/95 projects on Linux OS. It could be adapted, with some modifications, to other programming languages as well. Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. Make gets its knowledge of how to build your program from a file Gearscalled the makefile, which lists each of the non-source files and how to compute it from other files. When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program. The GNU `make' utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. Our examples show Fortran (F90/95) programs, but you can use `make' with any programming language (for example: C or C++) whose compiler can be run with a shell command. Indeed, 'make' is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change. Here is a basic description of the GNU make utility.

To prepare to use `make', you must write a file called the "makefile" that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files. Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. If you have several makefile files in the same project workspace / directory (default makefile file will be named: makefile), and some of these makefile files have different names, then you can execute them (specify which makefile you want to use) with the following command:

```
make -f name
```

where 'name' stands for the name of the makefile file that you want to use. Each of these makefiles can be completely different from each other. The `make' program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

Introduction to makefiles

You need a file called a "makefile" to tell `make' what to do. Most often, the makefile tells `make' how to compile and link a program. We will discuss only a simple makefiles that describe how to compile and link a fortran program which consists of several Fortran 90/95 source files. The makefile can also tell `make' how to run miscellaneous commands when explicitly asked (for example, to remove certain files as a clean-up operation). Let's say (for the sake of complexity) that one of these F90 source files is a module which holds global varibles, functions and subroutines that are seen from other subroutines (or functions) contained in other F90 source files. All F90 source files need to be compiled and linked together in order to produce the executable file which runs the program.

When `make' recompiles the program, each changed F90 source file must be recompiled as well. If a module file has changed (which holds global parameters accesed by other files), each F90 source file that includes this module file must be recompiled as well. Each compilation produces an object file corresponding to the source file (with extension: .o). Finally, if any source file has been recompiled, all the object files, whether newly made or obtained from previous compilations, must be linked together to produce the new executable file.

A simple makefile consists of "rules" with the following shape:

```
TARGET ... : PREREQUISITES ...
      COMMAND
      ...
```

A "target" is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as `clean'. A "prerequisite" is a file that is used as input to create the target. A target often depends on several files. A "command" is an action that `make' carries out. A rule may have more than one command, each on its own line.

Please Note:

You need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies commands for the target need not have prerequisites. For example, the rule containing the delete command associated with the target `clean' does not have prerequisites. A "rule", then, explains how and when to remake certain files which are the targets of the particular rule. `make' carries out the commands on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action.

A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this tutorial, but all fit the pattern more or less.

In order to demonstrate how to write makefiles, let us start with a simple example. We will use Intel Fortran Compiler (ifort) syntax for the compiler commands. Any other Fortran Compiler could be used instead (such as: G95 Fortran Compiler (g95), GCC Compiler Collection Fortran Compiler (gfortran), etc.). They all use the same syntax for compiling and linking fortran source files. First, we will start by compiling and linking (thus building) project for a simple example which is described next.

Let us say that we have four fortran source files (they all end with extension .f90), as follows:

```
main.f90
global.f90
function1.f90
subroutine1.f90
```

Source file: **main.f90** containes main program, which calls subsequent functions and subroutines. Each fortran program must have single main program with a key word: program. Source file: **global.f90** is module file containing several global variables (and/or functions (subroutines) definitions and/or

declarations). Let's say that this module is used from the main program and from function1, which is defined in file: function1.f90. Source file: **function1.f90** contains only one function, called: function1. This function is called from the main program. Source file: **subroutine1.f90** containes only one subroutine called: subroutine1. This subroutine is called from the main program. Both, function1 and subroutine1 are external to the main progam. Again, module is accessed from main program and function named: function1.

In order to build the main program, one needs to compile all four source files, and link them together to produce the executable file. Using the Intel Fortran Compiler (ifort) following commands could be issued (in the shell):

```
ifort global.f90 main.f90 function1.f90 subrouine1.f90
```

which should produce a file called: a.out. This is an executable file for this program. Compiler has compilled all four source files and produced respective object files (.o), and linked them together automatically. In the current directory one should find following additional files produced by the compiler: global.o, global.mod, main.o, function1.o and subroutine1.o. Files ending with .o are object files, while global.mod is a module file. This file takes the name of the module (here, both module and source file have the same name: global). This isn't always the case, but the principle remains the same.

If one wish to rename the executable file to something more meningfull than a.out, he should issue the following command (in the shell window):

```
ifort -o execname global.f90 main.f90 function1.f90 subrouine1.f90
```

which should produce a file called: execname. This is an executable file, which could be executed in the shell with the following command:

```
./execname
```

In order to compile single source file without automatic linking, one should issue (for example) the following command:

```
ifort -c function1.f90
```

which should produce a file called: function1.o. This is an object file for the source file: function1.f90. Each source file has a coresponding object file, which is produced as a product of a compilation process and is used in the linking process. In order to compile a source file which references (uses) module file (through the USE statement) one needs to compile the module file first. For example:

```
ifort -c global.f90
ifort -c main.f90
```

First command compiles global.f90 (without linking) and produces two files: global.o which is an object file, and global.mod (here, module is named global). This later file is a module file, and it is needed in the compilation process (second command) of the main.f90 (since it is using this module through the

USE statement), and needs to be available at the time of compilation. Hence, the order of the above commands must not be altered (reordered)!

Final linking of all compiled (module files) is carried out with the following command:

```
ifort global.o main.o function1.o subroutine1.o
```

which should produce: a.out, which is an executable file. If one wish to rename the executable file during the linking process, he should issue the following command (for example):

```
ifort -o execname global.o main.o function1.o subroutine1.o
```

or just by typing:

```
ifort -o execname *.o
```

which should produce executable file named: execname. Asterisk in the last command is a wildcard symbol which stands for: include all .o files in the current directory. Wildcard simbols could be used when issueing compiler and/or linker commands.

Please Note:

If one should wish to use different fortran compiler, such as g95 for example, he should just swap ifort with g95 in all above commands. Everything said earlier for Intel Fortran Compiler stands for g95 and gfortran as well. GCC Compiler Collection Fortran Compiler is activated with a gfortran command. All above examples could be carried out with this compiler as well, just swap ifort with gfortran.

When building fortran programs (which are consisted of several source code files, including one or more module files - all found in the same (working) directory) from the command line in (bash) shell, following procedure is the fastest way to produce the executable file.
First compile without linking every source file containing module declaration. Then issue the following command:

```
ifort -c *.f90
```

which compiles (without linking) all F90/95 source files found in the current (working) directory. This command produces necessary object files (.o files) for each source code file (.f90 file). After that issue the following command for linking all object files:

```
ifort -o execname *.o
```

which produces an executable file, named: execname.

Every time one of the source files change, one needs to compile it and link it against all other object files necessary to produce the executable file. If the number of source files is large, this could be difficult. Here comes the makefile utility to ease this process.

The same process of compiling and linking source files carryed out in the above example with issueing compiler commands in the shell command line could be easily accomplished with a makefile. Hence,

one needs to write a makefile only once and build program (compile + link = buld) with a command: **make** in the shell command line.

Here is a straightforward makefile that describes the way an executable file called: execname depends on four source files (one of which is a module file). This makefile is written for the above presented example.

```
# This is an commentary line in a makefile
# Start of the makefile
execname: global.o main.o function1.o subroutine1.o
    ifort -o execname global.o function1.o subroutine1.o
main.o
global.mod: global.o global.f90
    ifort -c global.f90
global.o: global.f90
    ifort -c global.f90
main.o: global.mod main.f90
    ifort -c main.f90
function1.o: global.mod function1.f90
    ifort -c function1.f90
subroutine1.o: subroutine1.f90
    ifort -c subroutine1.f90
clean:
    rm global.mod global.o main.o function1.o subroutine1.o
execname
# End of the makefile
```

This file should be saved under the directory where the source files are, and it should be given a following name: **makefile**. Long lines in a makefile could be continued with a following symbol: \. Commentary lines are presseded with a following symbol: #.To use this makefile to create the executable file called `execname', type:

```
Make
```

To use this makefile to delete the executable file and all the object files from the directory, type:

```
make clean
```

When a target is a file, it needs to be recompiled or relinked if any of its prerequisites change. In addition, any prerequisites that are themselves automatically generated should be updated first. A shell command follows each line that contains a target and prerequisites. These shell commands say how to update the target file. A tab character must come at the beginning of every command line to distinguish commands lines from other lines in the makefile. (Bear in mind that `make' does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All `make' does is execute the commands in the rule you have specified when the target file needs to be updated.)

The target `clean' is not a file, but merely the name of an action. Since you normally do not want to carry out the actions in this rule, `clean' is not a prerequisite of any other rule. Consequently, `make' never does anything with it unless you tell it specifically. Note that this rule not only is not a prerequisite,

it also does not have any prerequisites, so the only purpose of the rule is to run the specified commands. Targets that do not refer to files but are just actions are called "phony targets".

By default, `make' starts with the first target. This is called the "default goal". In the simple example of the previous section, the default goal is to update the executable program 'execname'; therefore, we put that rule first. Thus, when you give a command: make, `make' reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking `edit'; but before `make' can fully process this rule, it must process the rules for the files that `edit' depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each `.o' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist. The other rules are processed because their targets appear as prerequisites of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not processed, unless you tell `make' to do so (with a command such as `make clean').

After recompiling whichever object files need it, `make' decides whether to relink `execname'. This must be done if the file `execname' does not exist, or if any of the object files are newer than it. If an object file was just recompiled, it is now newer than `execname', so `execname' is relinked.

> Here are some usefull rules for creating simple makefiles. First line in the makefile should define the executable program, with listed all prerequisites (needed object and module files), followed by a compiler command in the next line. Tab character must be placed in front of the compiler command!
> After defining target (executable file), one should follow with definitions for the module file (.mod file) and then object file of the corresponding module file (.o of the module file). For the .mod file a prerequisites are corresponding .o and .f90 files. Tab character must be placed in front of the compiler command!
>
> The order just described above should always be followed!
>
> Now you can list all other object files (.o files) with their prerequisites, in any order, followed with a compiler directive to build that object file (preceded with a tab character). Order of this 'other' object files in a makefile is completely arbitrary.
>
> When listing an object file which uses module (such as: function1.o) corresponding module file (.mod file) must be specified as the prerequisite!

Rule 'clean' is arbitrary but usefull feature that should be implemented in all makefiles. It simply deletes (cleans) all object (and module) files from the current (working) directory, including (if you wish) an executable file.

In order to accelerate the process of writing makefiles, and to make them more readable one could use variables. Variables are also called macros. For example, in the above simple makefile example, the rule for makeing target: execname lists all object files as prerequisites. If a number of files in the project is large, this list could become quite long. It can be also seen from the above example that rule: clean also lists all this object files again.

In order to reduce this, a variable could be defined as follows:

```
objects = global.o main.o funkcija1.o subroutina1.o
```

and then use this variable when appropriate, as follows:

```
execname: $(objects)

    ifort -o execname $(objects)

clean:

    rm execname $(objects)
```

Another usefull way of using variables (macros) is to define the type / name of compiler which is used in the conrete makefile. For example, if you wish to use the above makefile with a g95 fortran compiler you would have to replace every single 'ifort' command with a 'g95' command, throughout the makefile. This can be rather difficult and error prone. Easier way is to define the compiler command (variable) at the beginig of the makefile, such as:

```
f90comp = ifort
```

The same makefile presented above should now look something like this:

```
# Start of the makefile
# Defining variables
objects = global.o main.o function1.o subroutine1.o
f90comp = ifort
# Makefile
execname: $(objects)
    $(f90comp) -o execname $(objects)
global.mod: global.o global.f90
    $(f90comp) -c global.f90
global.o: global.f90
    $(f90comp) -c global.f90
main.o: global.mod main.f90
    $(f90comp) -c main.f90
function1.o: global.mod function1.f90
    $(f90comp) -c function1.f90
subroutine1.o: subroutine1.f90
    $(f90comp) -c subroutine1.f90
# Cleaning everything
clean:
    rm global.mod execname
    rm $(objects)
# End of the makefile
```

If you now wish to change compiler command from ifort to for example g95 all you need to do is alter the line where compiler command is defined (under variables), as follows:

```
f90comp = g95
```

and that is it! This same makefile can be now used with a g95 fortran compiler, with the 'make' command. Variables are defined at the begining of the makefile! Usefull variable which can be defined for the makefiles is one which defines compiler options. In this way, various compiler optiones (switches) could be activated / deactivated in a simple manner. For example, one can define the following variable:

```
switch = -O3
```

which represents a compiler optimization option for the Intel Fortran Compiler on Linux. With this new variable, above presented makefile looks like this:

```
# Start of the makefile
# Defining variables
objects = global.o main.o function1.o subroutine1.o
f90comp = ifort
switch = -O3
# Makefile
execname: $(objects)
    $(f90comp) -o execname $(switch) $(objects)
global.mod: global.o global.f90
    $(f90comp) -c $(switch) global.f90
global.o: global.f90
    $(f90comp) -c $(switch) global.f90
main.o: global.mod main.f90
    $(f90comp) -c $(switch) main.f90
function1.o: global.mod function1.f90
    $(f90comp) -c $(switch) function1.f90
subroutine1.o: subroutine1.f90
    $(f90comp) -c $(switch) subroutine1.f90
# Cleaning everything
clean:
    rm global.mod
    rm $(objects)
# End of the makefile
```

If one wants to remove optimization or compile with different optimization options all he/she needs to do is redefine that variable at the beginning of the makefile.

Inference rules generalize the build process so you don't have to give an explicit rule for each target. As an example, compiling Fortran 90 source (.f90 files) into object files (.o files) is a common occurrence as can be seen from the above example. Rather than requiring a statement that each .o file depends on a like-named .f90 file, make uses an inference rule to infer that dependency. The source determined by an inference rule is called the inferred source. Inference rules are rules distinguished by the use of the character “%” in the dependency / rule line of the makefile. Hence, one can give make a set of rules for creating files with a certain suffix from files with the same (or approximatelly the same) root file name, but a different suffix. For example, the following lines in the makefile:

```
%.o: %.f90

    $(f90comp) -c $(switch) $<
```

tells make that all .o files are created from the corresponding **.f90** files. The command that followed will recompile any .f90 file if it is newer than the corresponding .o file. Hence, this line uses the make internal macro **$<**, which translates to "any dependency that is more recent than its corresponding target." This internal macro can only be used in suffix rules. Exceptions to the suffix rule can be stated explicitly. By using this newly introduced suffix rule, above described makefile might look something like this:

```
# Start of the makefile
# Defining variables
objects = global.o main.o function1.o subroutine1.o
f90comp = ifort
switch = -O3
# Makefile
execname: $(objects)
    $(f90comp) -o execname $(switch) $(objects)
global.mod: global.o global.f90
    $(f90comp) -c $(switch) global.f90
global.o: global.f90
    $(f90comp) -c $(switch) global.f90
main.o: global.mod main.f90
    $(f90comp) -c $(switch) main.f90
function1.o: global.mod function1.f90
    $(f90comp) -c $(switch) function1.f90
%.o: %.f90
    $(f90comp) -c $(switch) $<
# Cleaning everything
clean:
    rm global.mod
    rm $(objects)
# End of the makefile
```

There isn't much need for this suffix rule in the above example, due to it's simplicity. But, by this rule, every source code file (in the project workspace) which doesn't use module file will be included in the makefile. In this way, makefiles for very large projects (with large number of source code files) could be easily generated / written. In this way, by using suffix rules, one doesn't need to write makefile rule for each source code file. One suffix rule with $< macro will define makefile rule for the majority of source code files. Additional rules for those source code files which use modules are written separately. Rule which states that module file needs to be compiled before source code file (which uses that module file) still holds.

This was a very brief introductory tutorial to creating simple makefiles, that could be used in everyday programming, in order to accelerate the process of building fortran programs in Linux environment. Makefiles are very usefull tool for building large programs consisted of dozens of source code files (including module files). Everything said here implyed that you were using Fortran as a programming language. Another simple tutorial for creating makefiles for Fortran language can be found [redacted]. Simmilar makefiles could be created for C programming language, as well. A more complex tutorial for creating makefiles can be found [redacted] (it is written for the C language).