Session – 10 : Introduction to the make command
29-08-2019, 09:00 – 10:00 Hrs, RJN 302

[1] The make command

The `make` utility is meant to help with conditional and file wise compiling of large codes. It is designed in a generic manner and can be used for regular book keeping tasks too. By default, make will look for a file named Makefile in the current directory for instructions. This can be overridden and `make` can be asked to use instructions from another file, say, `mymakefile.txt` using the command "`make -f mymakefile.txt`"

The rules to recompile are listed in the Makefile as follows:

```
target : prerequisites
     recipe
```

Each line of recipe should have tab in the beginning. Override this using the variable ".RECIPEPREFIX".

Backslash "\" is used to continue a line.

The first target is what the make starts with. The variable ".DEFAULT_GOAL" can override this.

Use variables to hold list of objective functions to avoid spelling errors.

```
objects = main.o test.o output.o
mytarget : $(objects)
     gcc -o mytarget $(objects)
.PHONY : clean
clean :
     rm mytarget $(objects)
```

This ".PHONY" line asks make to ignore a file called "clean" in the current directory and perform the action "clean" as expected.

Makefiles contain the following:

Explicit Rule	When and how to make targets
Implicit Rule	When and how to make a class of files
Variable Definition	Text string value for a variable that can be substituted later
Directive	Reading another makefile, Deciding whether or not to use part of a makefile, Defining a variable from a verbatim string containing multiple lines.
Comment	Starts with # character.

The “include” directive allows for reading one or more other makefiles before proceeding.

Look at the folder Temp2D kept in the course website for illustration.

[2] Using shortcuts

You can use short cuts or aliases to save lot of editing to make your Makefile customized for different compilers or options.

The following contents of a Makefile that can be used with the code “qtree” shared on the moodle page illustrates these features.

```
cc = g++
CC = g++
CFLAGS = -g3 -ggdb -O

# Modify the following paths to suit your machine
CODEBASE=/home/blah/qtree
INSTALLDIR=$(CODEBASE)/run
BACKUPDIR=$(CODEBASE)/backup

default: main.o qarray.o qtree.o user.o
	$(cc) $(CFLAGS) main.o qarray.o qtree.o user.o -o qtree.e

main.o: main.cpp protos.hpp
	$(cc) $(CFLAGS) -c main.cpp -o main.o

qarray.o: qarray.cpp qarray.hpp
	$(cc) $(CFLAGS) -c qarray.cpp -o qarray.o

qtree.o: qtree.cpp qtree.hpp
	$(cc) $(CFLAGS) -c qtree.cpp -o qtree.o

user.o: user.cpp user.hpp
	$(cc) $(CFLAGS) -c user.cpp -o user.o

install:
	/bin/mv qtree.e $(INSTALLDIR)

clean:
	/bin/rm -f *.o qtree.e *.gch

DATESTAMP:=$(shell date +"%Y-%m-%d_%H.%M")
TARBALL=codebackup_$(DATESTAMP).tar

tar:
	# ------- making a tarball ------------
	@echo "Backing up with datestamp: $(DATESTAMP)";
	@echo "Tarball name: $(TARBALL)";
	tar -cvf $(TARBALL) *.cpp *.hpp readme.txt Makefile
	@ls -l $(TARBALL)
	/bin/mv $(TARBALL) $(BACKUPDIR)
```

```
	# ------- done moving tarball ------------

# ------------ end of Makefile ----------------
```

Homework:

[1] Take one of your old codes, split the code into separate files, one for each function. Create a makefile and test the recompilation.

[2] Create a makefile that uses a pattern for files rather than explicit listing of each of the files.

[3] Create a Makefile that does simple book keeping tasks such as the following:

Command to be made to work	**Outcome to be made sure**
`make tmpclean`	Remove files older than a day from /tmp folder
`make backup`	Copy source codes to a backup folder using the date stamp in the filename itself.
`make diff`	Using the files in the backup folder, show the difference in the files that are modified recently.