Make: an automation building tool

(PHY1610 – Lecture 4)

Ramses van Zon & Marcelo Ponce

SciNet HPC Consortium/Physics Department
University of Toronto

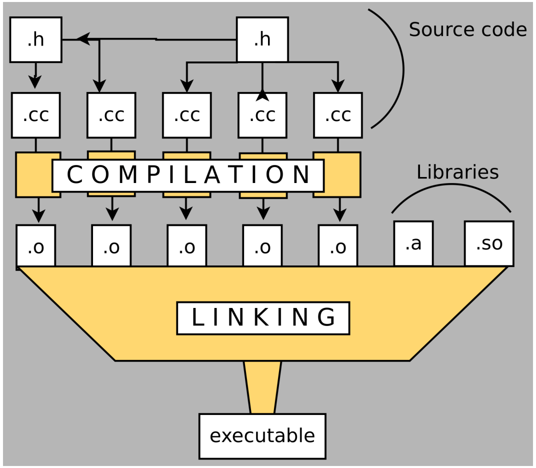
January 21st, 2021

Today's class

Today we will discuss the following topics:

- Review of Modularity and Automation
- The '`make`' command.
- `make` makefiles/rules/patterns/...
- `make` examples.

Modularity and Automatization



- header (.h) files
- implementation (.cc/.cpp/.cxx) files
- objects (.o) files – generated by the compilation stage
- library files (.a/.so/...)
- an executable – generated in the linking stage

* **make** can help to automize and generate each of the steps in the process

make: Origin

The typical compilation of a large program is a two-step process:

1. First individually compile all .cpp files, but not the .h (header) files, to generate .o (library) files.
2. Link all of the .o files together, including external .so and .a (shared-object and static library files), to generate an executable.

However, it can get complicated and redundant:

- you need to keep track of what depends upon what.
- you need to retype in the entire compilation command every time you need to recompile.
- It's easy to forget all of your compiler flags from one day to the next, as well as the location of external libraries.

It's better to keep all of this information contained in a single file.
This is where the '`make`' program enters the picture.

make: Generalities

`make` is a program that is used to build programs from multiple .cpp, .h, .o, and other files.

- `make` is a very general framework that is used to compile code, of any type.
- `make` takes a 'Makefile' as its input, which specifies what to do, and how.
- The Makefile contains variables, rules and dependencies.
- The Makefile specifies executables, compiler flags, library locations, ...
- It is a crutial component of *Professional Software Development*
- Besides building programs, `make` can be used to manage any project where some files must be updated automatically from others whenever the others change (eg. papers, ...)
- GNU make refs

make: basic usage

- Make is invoked with a list of target file names to build as *command-line arguments*,

```
$ make [TARGET ...]
```

- Without arguments, Make builds the first target that appears in its makefile, which is traditionally a symbolic "phony" target named ALL.
- Make decides whether a target needs to be regenerated by comparing file modification times (timestamp).
  This solves the problem of avoiding the building of files which are already up to date, but be aware that sometimes it may fail...
- Make takes many command-line arguments (`make --help`), and can also tell you the available targets in a *makefile*,

```
$ make --help
```

Makefiles

- Make searches the current directory for the makefile to use, (eg. `makefile`, `Makefile`, `GNUmakefile`) and then runs the specified (or default) target(s) from (only) that file
- It is possible to specify a different "makefile" by using a '`-f`' flag, eg.

```
$ make -f myMakefile [TARGET ...]
```

- the makefile is a plain-text file, with a particular structure
- it may include rules and even use commands from the shell

Rules

- A makefile consists of rules.
- Each rule begins with a textual dependency line which defines a target followed by a colon (:) and optionally an enumeration of components (files or other targets) on which the target depends.
  Eg. a target is a file to be created or updated.

```
TARGET: dependencies...
    [commnad 1]
    :
    [commnad n]
```

```
TARGET1 [TARGET2 ...]: dep1 dep2 ...
    [commnad 1]
    :
    [commnad n]
```

- The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon).
- It is common to refer to components as prerequisites of the target.
- each command-line must start with a ⇆ TAB , to be recognized as a command

```
Makefile:3:  *** missing separator.  Stop.
```

Rules – commands

- Each command is executed by a separate shell or command-line interpreter instance.
- backslash \ can be used to have commands executed by the same shell, it represents line-continuation
- commands can be separated by ;
- comments are included using #

```
# target:  list - List source files
list:
    # Won't work, each cmd is in separate shell
    cd src
    ls
    # Correct, continuation of the same shell
    cd src; \
    ls
```

- an @, results in the command not to be printed to standard ouput

```
# example of a simple makefile
hello:  ; @echo ``hello''
```

```
hello:
    @echo ``hello''
```

- A rule may have no command lines def. The dependency line can consist solely of components that refer to targets.

```
# example of a makefile, with
# multiple rules concatenated
realclean:  clean distclean
        ⋮
clean:  ...
        ⋮
distclean:  ...
```

Macros & Variables

- Macros are usually referred to as variables when they hold simple string definitions, like "`CXX = g++`".
- Macros in makefiles may be overridden in the command-line arguments passed to the Make utility.
- Macros allow users to specify the programs invoked and other custom behavior during the build process. For example, the macro "`CXX`" is frequently used in makefiles to refer to the location of a C compiler

```
MACRO = definition
```

```
PACKAGE = package
VERSION = ` date +"%Y.%m%d" `
ARCHIVE = $(PACKAGE)-$(VERSION)
dist:
    # Notice that only now macros are
    # expanded for shell to interpret:
    # tar -cf package-`date +"%Y%m%d"`.tar

    tar -cf $(ARCHIVE).tar .
```

- Environment variables are also available as macros.

make is sort of two languages in one

- The first language describes dependency graphs consisting of targets and prerequisites.
- The second language is a macro language for performing textual substitution.

Variables

Variables

A variable begins with a `$` and is enclosed within parentheses `(...)` or braces `{...}`.
Single character variables do not need the parentheses.
Egs. `$(CC)`, `$(CC_FLAGS)`, `$@`, `$^`.

Automatic Variables

- `$@`: the target filename
- `$*`: the target filename without the file extension
- `$<`: the first prerequisite filename
- `$^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.
- `$+`: similar to `$^`, but includes duplicates
- `$?`: the names of ll prerequisites that are newer than the target, separated by spaces

Special Rules

Suffix Rules

have target with names in the form "`.FROM.TO`", and are used to launch actions based on file extension.

Eg.

```
.SUFFIXES: .txt .html
# From .html to .txt
.html.txt:
	lynx -dump $< > $@
```

```
$ make file.txt
lynx -dump file.html > file.txt
```

`$<` refers to the first prerequisite

`$@` refers to the target

Suffix rules cannot have any prerequisites of their own.

Special Rules

Pattern Rules

A pattern rule looks like an ordinary rule, except that its target contains exactly one character '%'.
The target is considered a pattern for matching file names: the '%' can match any substring of zero or more characters, while other characters match only themselves.
The prerequisites likewise use '%' to show how their names relate to the target name.

Eg.

```
# From .html to .txt
%.txt :  %.html
    lynx -dump $< > $@
```

Use `$<` to refer to the first dependency of the current rule.
Use `$@` to refer to the target of the current rule.
Use `$^` to refer to the dependencies of the current rule.

phony target

A target that does not correspond to a file or other object.
Phony targets are usually symbolic names for sequences of actions.

```
.PHONY: variables
variables:
	@echo TXT_FILES: $(TXT_FILES)
```

```
.PHONY: clean
clean:
	rm -f *.dat
	rm -f results.txt
```

Compilation and Linking

example.cc

```
// example using the GSL library
#include <iostream>
#include <gsl/gsl_sf_bessel.h>
int main() {
   double x = 5.0;
   double y = gsl_sf_bessel_J0(x);
   std::cout << "J0(" << x << ") = "
       << y << std::endl;
   return 0;
}
```

* Compilation

```
$ g++ -std=c++11 -I/usr/local/include -c example.c

$ export GSL_INC=/usr/local/include
$ g++ -std=c++11 -I${GSL_INC} -c example.c
```

* Linking with libraries

```
$ g++ -std=c+11 -L/usr/local/lib example.o -lgsl

$ export GSL_LIB=/usr/local/lib
$ g++ -std=c++11 -L${GSL_LIB} example.o -lgsl
```

Using SciNet's TEACH (NIAGARA)

Use the appropriate *module* (eg. `module load gsl/2.4`), and the corresponding environment variables: `${SCINET_GSL_ROOT}`
– (actually not needed!) `${SCINET_GSL_ROOT}/include` `${SCINET_GSL_ROOT}/lib`

Compiling with make

How does make work?

- A makefile 'rule' is a word followed by a colon (:).
- By default make will execute the first rule it encounters.
- After the colon are the dependencies of the rule.
- When make hits a dependency it goes and looks for it.
- When it runs out of rules for the dependencies, it checks the timestamps; if the dependency is newer than the rule the command is executed.

```
# This file is called Makefile
# for compiling a program using GSL

# Define the compiler to use.
CXX = g++

# Compiler and linker flags.
GSL_INC ?= .  ; GSL_LIB ?= .
CXXFLAGS = -I${GSL_INC} -O2
LDFLAGS = -L${GSL_LIB}
LDLIBS = -lgsl -lgslcblas

all: myprog

myprog: myprog.o
   ${CXX} -o myprog myprog.o ${LDFLAGS} ${LDLIBS}

myprog.o: myprog.cpp
   ${CXX} ${CXXFLAGS} -c -o myprog.o myprog.cpp
```

Compiling multiple source files with make

How does make work?

- `make` will only recompile those dependencies that have source files that are newer then the library, thus only the code you are working on is modified.
- The most annoying part of make: the indentation of the command after the rule is actually a 'tab' (↹), and it must be a tab.
- The \ symbol indicates a line-continuation.

```
# Makefile
CXX = g++
GSL_INC ?= .   ; GSL_LIB ?= .
CXXFLAGS = -I${GSL_INC} -O2
LDFLAGS = -L${GSL_LIB}
LDLIBS = -lgsl -lgslcblas

all: MyArray

MyArray: MyArray.o outputarray.o
	${CXX} -o MyArray MyArray.o \
	outputarray.o ${LDFLAGS} ${LDLIBS}

MyArray.o: MyArray.cpp outputarray.h
	${CXX} ${CXXFLAGS} -c -o MyArray.o MyArray.cpp

outputarray.o: outputarray.cpp
	${CXX} ${CXXFLAGS} -c -o outputarray.o outputarray.cpp
```

Put a 'clean' rule in your Makefile

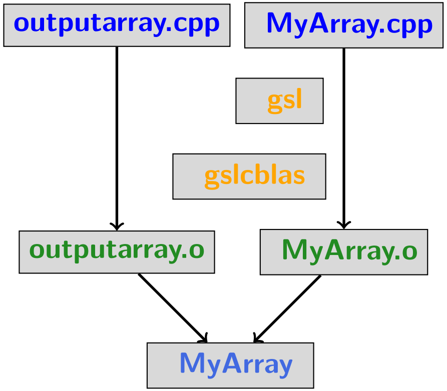
```
CXX = g++
GSL_INC ?= .  ; GSL_LIB ?= .
CXXFLAGS = -I${GSL_INC} -O2
LDFLAGS = -L${GSL_LIB}
LDLIBS = -lgsl -lgslcblas

MyArray: MyArray.o outputarray.o
   ${CXX} -o MyArray MyArray.o outputarray.o ${LDFLAGS}
   ${LDLIBS}

MyArray.o: MyArray.cpp outputarray.h
   ${CXX} ${CXXFLAGS} -c -o MyArray.o MyArray.cpp

outputarray.o: outputarray.cpp
   ${CXX} ${CXXFLAGS} -c -o outputarray.o outputarray.cpp

clean:
   rm -f MyArray.o outputarray.o MyArray
```



```
$ make clean
$ make
```

Parallel Execution

- Normally, `make` will execute only one recipe at a time, waiting for it to finish before executing the next
- However, the '`-j`' or '`--jobs`' option tells `make` to execute many recipes simultaneously.
- If the '`-j`' option is followed by an integer, this is the number of recipes to execute at once; this is called the number of job slots.
- If there is nothing looking like an integer after the '`-j`' option, there is no limit on the number of job slots.
- The default number of job slots is one, which means serial execution (one thing at a time).
- It is possible to **inhibit** parallelism in a particular makefile with the `.NOTPARALLEL` pseudo-target

Eg.

```
$ make -j 8
```

Summary

`make` utility

- automation tool
- mandatory in software development
- widely used for software installation
- not only used in software compilation, eg. latex-paper generation, ...
- there are several alternatives to make (eg. cmake, ...)

Best Practices on Scientific/Professional Software Development

- Modularity
- Automation Building Tool
- Version Control
- Defensive Programming
- Unit Testing

Example of a Makefile, for compiling a Latex-document

```
# Makefile for compiling a latex paper
NAME=manuscript
TARGET=$(NAME).pdf
SOURCE=$(NAME).tex

JUNK=.aux .bbl .blg .dvi .log .nav .out
     .ps .pdf .snm .tex.backup .tex.bak
     .toc Notes.bib

.PHONY: clean
```

```
$(TARGET): $(SOURCE)
        @pdflatex $(SOURCE)
        @bibtex $(NAME)
        @pdflatex $(SOURCE)
        @pdflatex $(SOURCE)

all: $(TARGET)

clean:
        @for ext in $(JUNK); do \
                rm —v $(NAME)$$ext; \
        done
```

Documenting Makefiles

This might result more useful, either when developing professional software or building pipelines.

`--help` will show available targets in a *makefile*,

```
$ make --help
```

It is possible to document the targets as well, eg

```
:
:
.PHONY : help
help :
	@echo "debug:  compile code with debugging flags."
	@echo "build:  compile code with default options."
	@echo "install:  install package."
	@echo "clean:  Remove auto-generated files."
```

```
$ make help
```

Self-documented Makefiles

It is possible to improve the way in which the targets are documented.

```
## debug:  compile code with debugging flags.
debug :  ...
⋮
## build:  compile code with default options.
build :  ...
⋮
## install:  install package.
install :  ...
⋮
## clean :  Remove auto-generated files.
.PHONY : clean
clean :
⋮
.PHONY : help
help :  Makefile
    @sed -n 's/^##//p' $<
```

```
$ make help
debug:  compile code with debugging flags.
build:  compile code with default options.
install:  install package.
clean:  Remove auto-generated files.
```