Programmazione Avanzata per il Calcolo Scientifico

Advanced Programming for Scientific Computing

Lab 02 - 20/03/2015

Carlo de Falco

Makefiles

Makefile example

Let's start off with the following three files, hellomake.c, hellofunc.c, and hellomake.h, which would represent a typical main program, some functional code in a separate file, and an include file, respectively.

```
// hellomake.cpp
// ------------------------

#include <hellomake.h>
#include <iostream>

int main()
{

  // call function from other file
  myPrintHelloMake ();
  std::cout
        << "Hello from C++ as well!"
        << std::endl;

  return (0);
}
```

```
// hellomake.h
// -------------------------

#ifdef __cplusplus
extern “C”{
#endif

void myPrintHelloMake (void);


#ifdef __cplusplus
}
#endif
```

```
// hellofunc.c
// -------------------------


#include <stdio.h>
#include <hellomake.h>

void myPrintHelloMake (void)
{
  printf("Hello makefiles!\n");
}
```

No Makefile

Normally, you would compile this collection of code by executing the following commands:

```
g++ -c hellomake.cpp -I.
gcc -c hellofunc.c -I.
g++ -o hellomake hellomake.o hellofunc.o
```

This compiles the two .c/.cpp files and names the executable hellomake.
The `-I.` is included so that gcc/g++ will look in the current directory (.) for the include file hellomake.h.

Without a makefile, if you are only making changes to one file, recompiling all of them every time is time-consuming and inefficient. So, it's time to see what we can do with a makefile.

Makefile I

The simplest makefile you could create would look something like:

```
hellomake: hellomake.cpp hellofunc.c
	gcc -c hellofunc.c -I.
	g++ -c hellomake.cpp -I.
	g++ -o hellomake hellomake.o hellofunc.o
```

One very important thing to note is that **there is a tab before the gcc/g++** commands in the makefile.
There **must** be a tab at the beginning of any command!

Put this rule into a file called Makefile hen type

make

on the command line.

With no arguments executes the first rule in the file.

The list of files on which the command depends is on the first line after the :,
make knows that the rule hellomake needs to be executed if any of those files change.

Makefile II

A slightly more efficient version:

```
CC=gcc
CXX=g++
CFLAGS=-I.
CXXFLAGS=-I.

hellomake: hellomake.o hellofunc.o
	$(CXX) -o hellomake hellomake.o hellofunc.o -I.
```

Constants CC, CXX and CFLAGS are special constants that communicate to make how we want to compile the files hellomake.cpp and hellofunc.c.

CC is the C compiler
CXX is the C++ compiler
CFLAGS/ is the list of flags to pass to the compilation command.

Building .o files follows an implicit rule

Makefile III

Let's take care of the header files:

```
CC=gcc
CXX=g++
CFLAGS=-I.
CXXFLAGS=-I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

%.o: %.cpp $(DEPS)
	$(CXX) -c -o $@ $< $(CXXFLAGS)

hellomake: hellomake.o hellofunc.o
	g++ -o hellomake hellomake.o hellofunc.o $(CXXFLAGS)
```

The macro DEPS is the set of .h files on which the .c/.cpp files depend.

"-o $@" says to put the output of the compilation in the file named on the left side of the :

"$<" is the first item in the dependencies list

Makefile IV

Let's take care of the header files:

```
CC=gcc
CXX=g++
CFLAGS=-I.
CXXFLAGS=-I.
LDFLAGS=
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

%.o: %.cpp $(DEPS)
	$(CXX) -c -o $@ $< $(CXXFLAGS)

hellomake: $(OBJ)
	g++ -o $@ $^ $(LDFLAGS)
```

$^ is the right side of the ":"

all of the object files should be listed as part of the macro OBJ

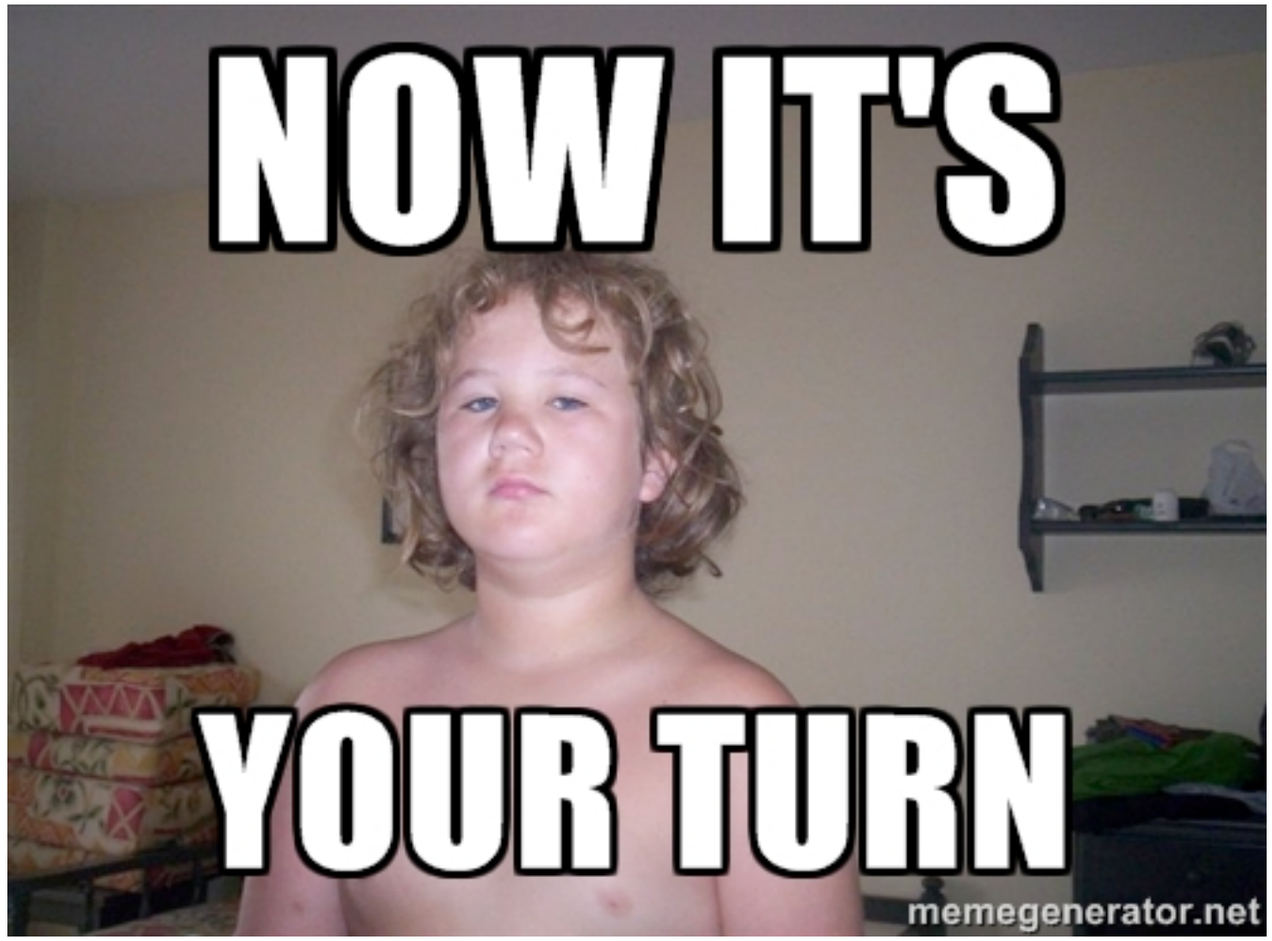
Makefile V

A more structured example:

```
IDIR =../include
CC=gcc
CXX=g++
CFLAGS=-I$(IDIR)
CXXFLAGS=-I$(IDIR)

ODIR=../obj
LDIR =../lib
BINDIR =../bin
LIBS=

_DEPS = hellomake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = hellomake.o hellofunc.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))

$(ODIR)/%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

$(ODIR)/%.o: %.cpp $(DEPS)
    $(CXX) -c -o $@ $< $(CXXFLAGS)

$(BINDIR)/hellomake: $(OBJ)
    $(CXX) -o $@ $^ $(CXXFLAGS) $(LIBS)

.PHONY: clean distclean

clean:
    $(RM) $(ODIR)/*.o *~ core $(INCDIR)/*~

distclean: clean
    $(RM) $(BINDIR)/hellomake
```

Makefile VI

Use variables defined by LMod!



CMake

CMake Tutorial

CMake is a **cross-platform**, open-source build system.

CMake is a family of tools designed to build, test and package software.

CMake is used to control the software compilation process using simple platform and compiler independent configuration files.

CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.

Let's follow the tutorial at http://www.cmake.org/cmake-tutorial/

Autotools

Autotools

The GNU build system, also known as the Autotools, is a suite of programming tools designed to assist in making source code packages portable to many **Unix-like systems**.

It consists of:

- GNU Autoconf
- GNU Automake
- GNU Libtool
- Gnulib