Make Tutorial

For : COP 3330.
Object oriented Programming (Using C++)

`http://www.compgeom.com/~piyush/teach/3330`

Piyush Kumar

Compiling programs

- Single source file code:
  - `g++ -g –Wall main.cpp –lm –o main`
- Multiple sources [a,b]
  - `g++ -g –c –Wall a.cpp`
  - `g++ -g –c -Wall b.cpp`
  - `g++ -g –o main a.o b.o -lm`

Compiler flags

- `-c`
  Separate compilation flag. Will produce a ``.o'' file only, without linking.
- `-g`
  The application will have full debug capabilities, but almost no optimization will be performed on the application, leading to lower performance.
- `-Wall`
  Enable all warnings.
- `-o filename`
  Write output to file.

For more options: “man g++”

Compilation

- Multi stage process
  - `g++  -g main.cpp –lm –o main`
    - `g++ -g -c main.cpp  –o main.o`
    - `g++ -g main.o main`
  - Compiling and Linking
    - Compilation : Produces object code (`main.o`)
    - Linking : Produces executable by linking a collection of object files and libraries.

A Typical Software Project

- Has 10s to 100s of source files
- Multiple directories
- Multiple authors
- Flags : Differ with compilation
  - Debugging flags (`-g`)
  - Optimization flags (`-O6 –malign-double`)
  - Release Vs Test builds.
- Make: A tool to automate the build process.
  - Other cool build tools: CMake, automake/autoconf, scons.

Make

- Make is not tied to any particular language.
- Make figures out automatically which files it needs to update, based on which source files have changed.
- Make enables the end user to build and install your package without knowing the details of how that is done.

Make and Makefiles

- "`make`" command reads "`makefile`" in the current directory for instructions for the build process.
- If you want to give it a specific file for input, say `Makefile-1` use
  - `make –f Makefile-1`

An example

- Main.cpp → Uses functions from other source files and is the main program.
- Hello.cpp → Function definition.
- Sumof.cpp → Function definition.
- Functions.hpp → Function declarations.

Download code from: http://www.compgeom.com/~piyush/teach/3330/examples/makex.tar.gz
Use "`tar zxf makex.tar.gz`" to untar files.

An Example: Makefile-0

- Running Make
  - `make -f Makefile-0`

    `g++ main.cpp hello.cpp sumof.cpp -o hello`
  - `make`

    `<reads default: Makefile>`

An Example: Makefile-0

- The basic makefile is composed of "rules":

  target: prerequisites
  <tab> system command

  Other targets
  Or source files

- `Makefile-0`

```
all:
 <tab> g++ main.cpp hello.cpp sunof.cpp -o hello
```

Default target for makefiles
"`make –f Makefile-0`" and "`make –f Makefile-0 all`" are equivalent.

Targets, Prereqs and commands

- 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.

Put a tab before the command

Another Example: Makefile-1

- `Makefile-1`

```
all: hello

hello: main.o sumof.o hello.o
        g++ main.o sumof.o hello.o -o hello

main.o: main.cpp
        g++ -c main.cpp

sumof.o: sumof.cpp
        g++ -c sumof.cpp

hello.o: hello.cpp
        g++ -c hello.cpp

clean:
        rm -rf *o hello *.exe
```

Dependencies

Command

Prerequisites / Dependencies.

Wildcards.

"`make –f Makefile-1 clean`" cleans up your directory except source code.

Make

- Only makes out of date prerequisites.

```
hello: main.o sumof.o hello.o
    g++ main.o sumof.o hello.o -o hello
```

- How to decide whether “hello” is out of date?
  - It is out of date if it does not exist,

    or
  - if either main.o , sumof.o or hello.o are more recent than it.

If “hello” is out of date, make executes the command ‘g++ main.o …’

Another Example: Makefile-2

- `Makefile-2`

Comments in makefile.

Variables

Using Variables

```
# The variable CC will be the compiler to use.
CC=g++

# CFLAGS will be the options I'll pass to the compiler.
CFLAGS=-c -Wall -g

all: hello

hello: main.o sumof.o hello.o
	$(CC) main.o sumof.o hello.o -o hello

main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp

sumof.o: sumof.cpp
	$(CC) $(CFLAGS) sumof.cpp

hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp

clean:
	rm -rf *o hello *.exe
```

Makefile targets

- Expected targets in makefiles
  - `make all`
    - Compile everything.
  - `make install`
    - Install your software.
  - `make clean`
    - Clean intermediate files and executables.

Make: How does it work?

- make reads the makefile in the current directory and begins by processing the first rule. (in our case: all)
- but before make can fully process this rule, it must process the rules for the files that ‘all’ depends on, which in this case are the object files.
- Each of these files is processed according to its own rule.

Make: How does it work?

- For any rule, the recompilation must be done if the prerequisites are more recent than the target, or if the target/object file does not exist.

Make: Wrap up

- make –j2
  - Uses 2 processors for the build process.
- More info:
  - http://www.gnu.org/software/make/manual/html_node/index.html
  - man make
  - Example : http://www.compgeom.com/~piyush/teach/3330/examples/makex.tar.gz