Makefile

Home

Schedule

Assignments

Makefile(s) and the "make" command makes compiling easier.

Makefile

The instructions used by the "make" command.

Makefiles can range from very simple to quite complex.

Makefiles provide the instructions to easily and repeatedly compile your source code.

The most basic Makefile

The most "bare-bones" Makefile example.

Given C++ source code "hello_world.cpp":

```
UNIX> cd /data/csci274/Lectures/makefile

UNIX> mkdir simple

UNIX> cd simple

UNIX> vim hello_world.cpp
```

```
#include <iostream>
using namespace std;

int main()
{
  cout << "HELLO WORLD!\n";
}
```

We could compile our source code into an executable named "hello_world" with the following command:

```
g++ hello_world.cpp -o hello_world
```

This command may become tedious to repeat over and over again.

Instead, we can create the following Makefile and use the "make" command:

```
UNIX> vim Makefile
```

```
all:
    g++ hello_world.cpp -o hello_world
```

*** IMPORTANT: THERE IS A TAB BEFORE "g++" ***

```
UNIX> make
g++ hello_world.cpp -o hello_world

UNIX> ls
hello_world hello_world.cpp Makefile

UNIX> ./hello_world
HELLO WORLD!

UNIX>
```

The "all:" line can specify which targets to follow and command(s) to execute:

```
UNIX> vim Makefile
```

```
all: hello_world

hello_world:
    g++ hello_world.cpp -o hello_world
```

Now, the Makefile specifies to execute the commands starting with the "hello_world:" target.

```
UNIX> rm hello_world

UNIX> ls
hello_world.cpp Makefile

UNIX> make
g++ hello_world.cpp -o hello_world

UNIX> ls
hello_world hello_world.cpp Makefile
```

If the "hello_world" executable is already in the directory, then make will not execute the commands:

```
UNIX> make
make: Nothing to be done for `all'.

UNIX>
```

Dependencies

Dependencies help ensure successful and up-to-date compilation.

Dependencies are mainly used to specify files that must be present for the commands to execute successfully:

```
UNIX> vim Makefile
```

```
all: hello_world

hello_world: hello_world.cpp
    g++ hello_world.cpp -o hello_world
```

Now the "hello_world" target is dependent on "hello_world.cpp"

make still compiles the executable as before:

```
UNIX> rm hello_world

UNIX> ls
hello_world.cpp Makefile

UNIX> make
g++ hello_world.cpp -o hello_world

UNIX> ls
hello_world hello_world.cpp Makefile

UNIX> ./hello_world
HELLO WORLD!

UNIX>
```

make won't compile if the executable already exists:

```
UNIX> make
make: Nothing to be done for `all'.

UNIX>
```

However, if we update hello_world.cpp, make will recompile:

```
UNIX> touch hello_world.cpp

UNIX> make
g++ hello_world.cpp -o hello_world

UNIX>
```

Dependencies are very helpful when compiling complicated source code:

Given the following five source files:

```
UNIX> cd ..

UNIX> mkdir dependencies

UNIX> cd dependencies/

UNIX> vim driver.cpp
```

```
#include "functions.h"

int main()
{
  function1();
  function2();
  function3();
}
```

```
UNIX> vim functions.h
```

```
#include <iostream>
using namespace std;
#pragma once

void function1();
void function2();
void function3();
```

```
UNIX> vim file1.cpp
```

```
#include "functions.h"

void function1()
{
  cout << "FUNCTION1\n";
}
```

```
UNIX> vim file2.cpp
```

```
#include "functions.h"

void function2()
{
  cout << "FUNCTION2\n";
}
```

```
UNIX> vim file3.cpp
```

```
#include "functions.h"

void function3()
{
  cout << "FUNCTION3\n";
}
```

Command line compilation would be a multi-step process

```
UNIX> g++ -c driver.cpp

UNIX> g++ -c file1.cpp

UNIX> g++ -c file2.cpp

UNIX> g++ -c file3.cpp
```

```
UNIX> g++ driver.o file1.o file2.o file3.o -o myProgram

UNIX> ./myProgram
FUNCTION1
FUNCTION2
FUNCTION3

UNIX>
```

If we made any changes to the source, we'd have to redo at least two of the above commands

make and Makefiles make this much easier

We can create a Makefile that lists targets of all the necessary dependencies and commands:

```
UNIX> vim Makefile
```

```
all: myProgram

myProgram: driver.o file1.o file2.o file3.o
    g++ driver.o file1.o file2.o file3.o -o myProgram

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

file1.o: file1.cpp
    g++ -c file1.cpp

file2.o: file2.cpp
    g++ -c file2.cpp

file3.o: file3.cpp
    g++ -c file3.cpp
```

Now, all: tells make to start with the target for myProgram

The myProgram target is dependent on many object files: driver.o, file1.o, file2.o, file3.o

If all of the object files (.o) are present in the current directory, the command following myProgram will execute.

If not, then the commands following the targets for the missing .o files will execute accordingly.

The targets for each object file is dependent on the presence of the .cpp source code files.

```
UNIX> rm driver.o file1.o file2.o file3.o myProgram

UNIX> ls
driver.cpp file1.cpp file2.cpp file3.cpp functions.h Makefile

UNIX> make
g++ -c driver.cpp
g++ -c file1.cpp
g++ -c file2.cpp
```

```
g++ -c file3.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram

UNIX> ls
driver.cpp driver.o file1.cpp file1.o file2.cpp file2.o file3.cpp
file3.o functions.h Makefile myProgram

UNIX> ./myProgram
FUNCTION1
FUNCTION2
FUNCTION3

UNIX>
```

If we attempt to "make" again, the commands will not execute since nothing has changed:

```
UNIX> make
make: Nothing to be done for `all'.
```

If we modify any of the source code files, make will only recompily what's necessary:

```
UNIX> touch driver.cpp

UNIX> make
g++ -c driver.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram

UNIX> touch file2.cpp

UNIX> make
g++ -c file2.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram

UNIX>
```

If we delete any of the object files, make will recompile only what's necessary:

```
UNIX> rm file1.o file2.o

UNIX> make
g++ -c file1.cpp
g++ -c file2.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram

UNIX>
```

Multiple executables

make and Makefiles can compile multiple executables

```
UNIX> cd ..

UNIX> cp -r dependencies/ multiple_exe

UNIX> cd multiple_exe/

UNIX> rm driver.o file1.o file2.o file3.o myProgram

UNIX> cp ../simple/hello_world.cpp .
```

```
UNIX> ls
driver.cpp file1.cpp file2.cpp file3.cpp functions.h hello_world.cpp
Makefile

UNIX> vim Makefile
```

To compile multiple executables, we simply add a new target to the all: line.

```
all: myProgram hello_world

myProgram: driver.o file1.o file2.o file3.o
    g++ driver.o file1.o file2.o file3.o -o myProgram

hello_world:
    g++ hello_world.cpp -o hello_world

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

file1.o: file1.cpp
    g++ -c file1.cpp

file2.o: file2.cpp
    g++ -c file2.cpp

file3.o: file3.cpp
    g++ -c file3.cpp
```

Now, make will start with myProgram target, and move on to hello_world target.

```
UNIX> make
g++ -c driver.cpp
g++ -c file1.cpp
g++ -c file2.cpp
g++ -c file3.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram
g++ hello_world.cpp -o hello_world

UNIX> ./myProgram
FUNCTION1
FUNCTION2
FUNCTION3

UNIX> ./hello_world
HELLO WORLD!

UNIX>
```

To make only specific programs, specify the target name as a command line argument to make:

```
UNIX> rm driver.o file1.o file2.o file3.o myProgram hello_world

UNIX> make myProgram
g++ -c driver.cpp
g++ -c file1.cpp
g++ -c file2.cpp
g++ -c file3.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram

UNIX> make hello_world
```

```
g++ hello_world.cpp -o hello_world

UNIX>
```

Notice that the command line arguments match the targets in the Makefile.. (i.e., "myProgram" and "hello_world")

---

clean

Add a target to delete specific files

Often times we want to clean up a directory.

Makefiles provides us with a great way to do this.

Simply add a target for "clean:" and the appropriate command to the makefile:

```
UNIX> vim Makefile
```

```
all: myProgram hello_world

clean:
    rm driver.o file1.o file2.o file3.o myProgram hello_world

myProgram: driver.o file1.o file2.o file3.o
    g++ driver.o file1.o file2.o file3.o -o myProgram

hello_world:
    g++ hello_world.cpp -o hello_world

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

file1.o: file1.cpp
    g++ -c file1.cpp

file2.o: file2.cpp
    g++ -c file2.cpp

file3.o: file3.cpp
    g++ -c file3.cpp
```

Now, when we type "make clean", it will call the appropiate target and commands from the Makefile:

```
UNIX> ls
driver.cpp driver.o file1.cpp file1.o file2.cpp file2.o file3.cpp
file3.o functions.h hello_world hello_world.cpp Makefile myProgram

UNIX> make clean
rm driver.o file1.o file2.o file3.o myProgram hello_world

UNIX> ls
driver.cpp file1.cpp file2.cpp file3.cpp functions.h hello_world.cpp
Makefile

UNIX> make
```

```
g++ -c driver.cpp
g++ -c file1.cpp
g++ -c file2.cpp
g++ -c file3.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram
g++ hello_world.cpp -o hello_world

UNIX> ./hello_world
HELLO WORLD!

UNIX> ./myProgram
FUNCTION1
FUNCTION2
FUNCTION3

UNIX>
```

Question: Why don't we include "clean" in the "all:" line?

Makefile Comments

Comments (ignored by make) are lines that start with '#'

```
UNIX> vim Makefile
```

```
#this is a comment
all: myProgram hello_world

#this is another comment
clean:
    rm driver.o file1.o file2.o file3.o myProgram hello_world

myProgram: driver.o file1.o file2.o file3.o
    g++ driver.o file1.o file2.o file3.o -o myProgram

#this is yet another comment
hello_world:
    g++ hello_world.cpp -o hello_world

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

file1.o: file1.cpp
    g++ -c file1.cpp

file2.o: file2.cpp
    g++ -c file2.cpp

file3.o: file3.cpp
    g++ -c file3.cpp
```

Macros

Macros cut down on repeated text in the Makefile

Many commands in the Makefile are repeated, in which case Macros come in handy:

```
UNIX> cd ..

UNIX> cp -r multiple_exe/ macros

UNIX> cd macros/

UNIX> vim Makefile
```

```
OBJS = driver.o file1.o file2.o file3.o
EXES = myProgram hello_world
FLAGS = -c
CC = g++

all: $(EXES)

clean:
    rm $(OBJS) $(EXES)

myProgram: $(OBJS)
    $(CC) $(OBJS) -o myProgram

driver.o: driver.cpp
    $(CC) $(FLAGS) driver.cpp

file1.o: file1.cpp
    $(CC) $(FLAGS) file1.cpp

file2.o: file2.cpp
    $(CC) $(FLAGS) file2.cpp

file3.o: file3.cpp
    $(CC) $(FLAGS) file3.cpp

hello_world: hello_world.cpp
    $(CC) hello_world.cpp -o hello_world
```

It's now very easy, for example, to add compiler flags or new executables.

make still works:

```
UNIX> make clean
rm driver.o file1.o file2.o file3.o myProgram hello_world

UNIX> make
g++ -c driver.cpp
g++ -c file1.cpp
g++ -c file2.cpp
g++ -c file3.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram
g++ hello_world.cpp -o hello_world

UNIX> touch file1.cpp

UNIX> make
g++ -c file1.cpp
g++ driver.o file1.o file2.o file3.o -o myProgram

UNIX>
```