Makefile Tutorial

Makefile Getting Started

What is Make?

Make is a program that looks for a file called "makefile" or "Makefile", within the makefile are variables and things called dependencies. There are many things you can do with makefiles, if all you've ever done with makefiles is compile C or C++ then you are missing out. Pretty much anything that needs to be compiled (postscript, java, Fortran), can utilize makefiles.

Format of Makefiles -- Variables

First, lets talk about the simpler of the two ideas in makefiles, variables. Variable definitions are in the following format:

```
VARNAME = Value
```

So lets say i want to use a variable to set what compiler i'm going to use. This is helpful b/c you may want to switch from cc to gcc or to g++. We would have the following line in our makefile

```
CC = gcc
```

this assigns the variable CC to the string "gcc". To expand variables, use the following form:

```
${VARNAME}
```

So to expand our CC variable we would say:

```
${CC}
```

Format of Makefiles -- Dependencies

Dependencies are the heart of makefiles. Without them nothing would work. Dependencies have the following form:

```
dependecy1: dependencyA dependencyB ... dependencyN
    command for dependency1
```

Very Important :

The commands underneath the dependencies must have a tab before them. This lets make know it is a command and not some kind of crazy dependency.

So how do we read what was just written above? Dependency1 is dependent upon dependencyA, and dependencyB, up to however many dependencies you have. The program make, looks at dependency1 and sees it is dependent upon dependencyA, so then it looks later in the makefile for how dependencyA is defined. Lets make a sample makefile:

```
myprogram: mainprog.cc myclass.cc
    gcc mainprog.cc myclass.cc
```

That is probably one of the simplest makefiles that could be made. When you type make, it automatically knows you want to compile the 'myprogram' dependency (because it is the first dependency it found in the makefile). It then looks at mainprog.cc and sees when the last time you changed it, if it has been updated since last you typed 'make' then it will run the 'gcc mainprog.cc ..." line. If not, then it will look at myclass.cc, if it has been edited then it will execute the 'gcc mainprog.cc ..." line, otherwise it will not do anything for this rule. Another example might be easier to see for dependencies, lets say you have a class, and more than one file depends upon objects of that class type. Then it makes sense to create a .o file of that class and compile it in with the file. Here is a sample of what I'm talking about

```
myprogram: mainprog.cc subfile1.o myclass.o
    gcc mainprog.cc subfile1.o myclass.o

subfile1.o: subfile1.cc myclass.o
    gcc -c mainprog.cc subfile1.o myclass.o

myclass.o: myclass.cc
    gcc -c myclass.cc
```

Notice the dependencies here, when 'make' is run, the dependency called "myprogram" is run first, and then the dependency subfile1.o is found, and then the myclass.o dependency is foudn withint subfile1.o. So myclass.cc is compiled, then subfile1.o is compiled, then we are finally back up to the myprogram dependency. Finally all three dependencies are compiled in to create a.out.

Tying variables and dependencies together

You can use variables within your dependencies very easily. Lets take that last example, and use two variables, one for the compiler we are going to use, and another for the options to that compiler. This makes life a lot simpler if say we want to turn off debugging for all the code, or turn it on, whichever is preferred. Here is an example tying it together.

```
COMPILER = gcc
CCFLAGS = -g
myprogram: mainprog.cc subfile1.o myclass.o
    ${COMPILER} ${CCFLAGS} mainprog.cc subfile1.o myclass.o

subfile1.o: subfile1.cc myclass.o
    ${COMPILER} ${CCFLAGS} -c mainprog.cc subfile1.o myclass.o

myclass.o: myclass.cc
    ${COMPILER} ${CCFLAGS} -c myclass.cc
```

notice how clean this is? It is easy to add extra options and libraries to your compiler if need be. Life is a whole lot simpler with makefiles, for a final addition to our makefile lets make something that cleans out all our code, starting with what we had before, we simply add a new dependency, named "clean". The addition goes as follows:

```
COMPILER = gcc
CCFLAGS = -g
myprogram: mainprog.cc subfile1.o myclass.o
    ${COMPILER} ${CCFLAGS} mainprog.cc subfile1.o myclass.o

subfile1.o: subfile1.cc myclass.o
    ${COMPILER} ${CCFLAGS} -c mainprog.cc subfile1.o myclass.o

myclass.o: myclass.cc
    ${COMPILER} ${CCFLAGS} -c myclass.cc

clean:
    rm -rf *.o a.out
```

So when we type 'make clean' at the command line, it will remove all our .o files and the executable that we have.

Conclusions

Makefiles are very very helpful when dealing with projects. The first thing you should do when making a project is create a makefile, it makes your life a lot

easier. They are very powerful things, if you have any questions feel free to email me at bhumphre@cs.ohiou.edu [1] **have fun!**
Last modified: Mon Oct 16 13:21:58 EDT 2000

1. emailto:bhumphre@cs.ohiou.edu