Understanding Make

Why Make?

Its easy and efficient to build small projects with a compile command. For example,

```
cc myapp.c -o myapp.
```

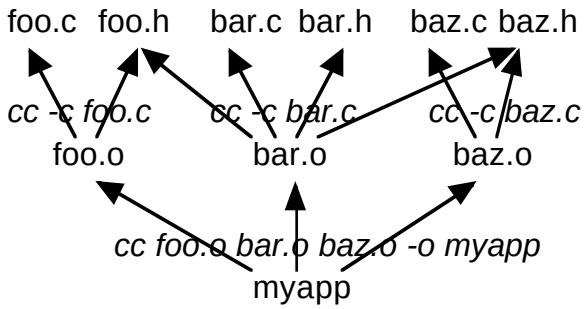
However, this method becomes very inefficient for large projects such as the ones you will be building in this class. For example, running

```
cc foo.c bar.c baz.c -o myapp
```

when `baz.c` changes means you are wasting time compiling the other files. Ideally, you would like to only compile `baz.c` and then link it with the other object files. You could do this yourself, but *make* lets you automate it. The goal of *make* is to build your project with the minimum amount of work.

Basic Idea

You supply *make* with a file (whose default name is "`Makefile`") which describes the dependencies between files in your project and a method for satisfying each dependency. These dependencies form a DAG - for example:



Now, when *make* is run, for each dependency, if the target file is older than the file it depends on, it will execute the method to bring the target file up to date. For example, suppose we change `foo.h`. *Make* will recognize that `foo.o` and `bar.o` depend on it and will recompile them. Next, it will relink `myapp`.

Simple Makefile

A makefile consists of one or more rules which have the form:

```
target : source(s)
[TAB]command
[TAB]command
```

The first character of each command line must be a TAB. The makefile corresponding to the above graph is

```
myapp: foo.o bar.o baz.o
      cc foo.o bar.o baz.o -o myapp

foo.o: foo.c foo.h
      cc -c foo.c

bar.o: bar.c bar.h foo.h  baz.h
      cc -c bar.c

baz.o: baz.c baz.h
      cc -c baz.c
```

Comments

Any line beginning with a '#' is ignored by *make*.

Makedepend

One headache with makefiles is making sure that you have specified dependencies to header files correctly. *Makedepend* is a tool that will do this for you automatically. You run *makedepend* on all your source files:

```
makedepend foo.c bar.c baz.c
```

and it will add the correct dependencies to Makefile. You can use the `-MM` option of *gcc* to do the same thing.

Macros

Makefiles can have macro definitions and uses. For example, with

```
CC    = gcc
CCOPT = -g -DDEBUG  -DPRINT
#CCOPT = -O2

foo.o: foo.c foo.h
      $(CC) $(CCOPT) -c foo.c
```

foo.c will be compiled for debugging or with

optimization depending on which `CCOPT` is uncommented.

Macros definitions can also be modified when they are used. For example,

```
OBJECTS = foo.o bar.o baz.o

dep:
      makedepend $(OBJECTS:.o=.c)
```

will cause *makedepend* to be called on `foo.c`, `bar.c` and `baz.c` when the `dep` target is made.

Suffix Rules

Often, a project has many rules that have common commands applied to files with the same suffixes. For example, each of the .o files in our example depend on its parent .c file and is compiled with the same command. We can replace this with a suffix rule:

```
.c.o :
      $(CC) $(CCOPT) -c $*.c -o $@
```

`$*` is a special macro for the prefix the two files share and `$@` contains the target name.

Default Rules

Make has a lot of built in defaults that are used when a user-defined rule can't be found. For example, it can infer that `foo.o` depends on `foo.c` and use a generic C compilation rule to update `foo.o`. In general, AVOID USING THE DEFAULT RULES.

A Menagerie of Makes

There are many different Makes with widely varying features. Perhaps the most popular alternate make is GNU Make, which has the advantage of having a manual freely available. Some makes exploit the parallelism of the dependency graph to distribute the make across a number of workstations.

To Learn More

man make
O'Reilly books in bookstore

A Larger Example

```
CC    = gcc
#CC    = cc
CPP   = g++
INC   = -ILEDA/incl
LIB   = -LLEDA
CCOPT = -g -DDEBUG -DSPACEMONITOR\
            $(INC) $(LIB)
#CCOPT  = -O2 $(INC) $(LIB)
CPPOPT= $(CCOPT)

GENERALS  = cache.o disthandler.o \
 internals.o dispatcher.o util.o \
 builder.o group.o relation.o \
 errprint.o diffmaprle.o aapair.o \
 aablock.o stdrel_llb.o \
 stdrel_sortseqpair.o

DISPLAY   = display.o
INTERNALS = HPF.o

DISPLIBS  = -lP -lG -lL -lWx -lX11 -lm

all: fung

fung: libdist.a libfxtimers.a fung.o \
  $(DISPLAY)
      $(CPP) $(CPPOPT) fung.o \
        libdist.a libfxtimers.a \
        $(DISPLAY) $(DISPLIBS) -o fung


libdist.a : $(GENERALS) $(INTERNALS)
      ar ruv libdist.a $(GENERALS) \
          $(INTERNALS)

$(DISPLAY): display.C
      $(CPP) $(CPPOPT) -c display.C \
         -o $(DISPLAY)

.c.o :
      $(CC) $(CCOPT) -c $*.c -o $@

dep:
      makedepend $(INC) \
        $(GENERALS:.o=.c) \
        $(INTERNALS:.o=.c) \
        $(DISPLAY:.o=.C)
```