Using make and writing Makefiles

make is a Unix tool to simplify building program executables from many modules. make reads in rules (specified as a list of target entries) from a user created Makefile. make will only re-build things that need to be re-built (object or executables that depend on files that have been modified since the last time the objects or executables were built).

GNU make Manual. A complete reference for writing makefiles from simple to advanced features.

Also, CMake, is a cross platform build system. Among other things, cmake will generate makefiles for you. It is particularly useful for large projects, for builds that use lots of libraries, and for dealing with platform-specific compilation issues. It automatically generates (often hard to read and debug) makefiles for different platforms. For small projects, writing makefiles by hand is likely easier. See Andy Danner's Cmake documentation for more information on cmake.

Using make

1. Create a Makefile listing the rules for building the executable the file should be named 'Makefile' or 'makefile'. This only has to be done once, except when new modules are added to the program, the Makefile must be updated to add new module dependencies to existing rules and to add new rules to build the new modules.
2. After editing program file(s), rebuild the executable by typing make:

```
% make

A specific target in the Makefile can be executed by typing:
% make target_label

For example, to execute the rm commands in the example makefile below, type:
% make clean
```

Creating a Makefile

A Makefile typically starts with some variable definitions which are then followed by a set of target entries for building specific targets (typically .o & executable files in C and C++, and .class files in Java) or executing a set of command associated with a target label.

The following is the generic target entry form:

```
# comment
# (note: the <tab> in the command line is necessary for make to work)
target:  dependency1 dependency2 ...
      <tab> command

for example:
#
# target entry to build program executable from program and mylib
# object files
#
program: program.o mylib.o
        gcc -o program program.o mylib.o
```

Example simple Makefiles for a C (or C++)

The most simple Makefile for compiling a C (or C++) program from a single .c file, with make and make clean rules, looks something like this (remember to add a TAB character before the command part):

```
  # build an executable named myprog from myprog.c
  all: myprog.c
          gcc -g -Wall -o myprog myprog.c

  clean:
          $(RM) myprog
```

a slightly more generic simple makefile

A slightly more generic version of the makefile above, uses makefile variables (just change the variable definitions to build differnet executables or with different compilers or compiler flags):

```
  # the compiler: gcc for C program, define as g++ for C++
  CC = gcc

  # compiler flags:
  #  -g    adds debugging information to the executable file
  #  -Wall turns on most, but not all, compiler warnings
  CFLAGS  = -g -Wall

  # the build target executable:
  TARGET = myprog

  all: $(TARGET)

  $(TARGET): $(TARGET).c
        $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c

  clean:
        $(RM) $(TARGET)
```

An example of building an executable from multiple .o files:

```
#
# This is an example Makefile for a countwords program.  This
# program uses both the scanner module and a counter module.
# Typing 'make' or 'make count' will create the executable file.
#

# define some Makefile variables for the compiler and compiler flags
# to use Makefile variables later in the Makefile: $()
#
#  -g    adds debugging information to the executable file
#  -Wall turns on most, but not all, compiler warnings
#
# for C++ define  CC = g++
CC = gcc
CFLAGS  = -g -Wall

# typing 'make' will invoke the first target entry in the file
# (in this case the default target entry)
# you can name this target entry anything, but "default" or "all"
# are the most commonly used names by convention
#
default: count

# To create the executable file count we need the object files
# countwords.o, counter.o, and scanner.o:
#
count:  countwords.o counter.o scanner.o
        $(CC) $(CFLAGS) -o count countwords.o counter.o scanner.o

# To create the object file countwords.o, we need the source
# files countwords.c, scanner.h, and counter.h:
#
countwords.o:  countwords.c scanner.h counter.h
        $(CC) $(CFLAGS) -c countwords.c

# To create the object file counter.o, we need the source files
# counter.c and counter.h:
#
counter.o:  counter.c counter.h
        $(CC) $(CFLAGS) -c counter.c

# To create the object file scanner.o, we need the source files
# scanner.c and scanner.h:
#
scanner.o:  scanner.c scanner.h
        $(CC) $(CFLAGS) -c scanner.c

# To start over from scratch, type 'make clean'.  This
# removes the executable file, as well as old .o object
# files and *~ backup files:
#
clean:
        $(RM) count *.o *~
```

Another makefile (using makedepend and more advanced make syntax)

This is an easier to use and modify makefile, but it is slightly more difficult to read than the simple one:

```
#
# 'make depend' uses makedepend to automatically generate dependencies
#               (dependencies are added to end of Makefile)
# 'make'        build executable file 'mycc'
# 'make clean'  removes all .o and executable files
#

# define the C compiler to use
CC = gcc

# define any compile-time flags
CFLAGS = -Wall -g

# define any directories containing header files other than /usr/include
#
INCLUDES = -I/home/newhall/include  -I../include

# define library paths in addition to /usr/lib
#   if I wanted to include libraries not in /usr/lib I'd specify
#   their path using -Lpath, something like:
LFLAGS = -L/home/newhall/lib  -L../lib

# define any libraries to link into executable:
#   if I want to link in libraries (libx.so or libx.a) I use the -llibname
#   option, something like (this will link in libmylib.so and libm.so:
LIBS = -lmylib -lm

# define the C source files
SRCS = emitter.c error.c init.c lexer.c main.c symbol.c parser.c

# define the C object files
#
# This uses Suffix Replacement within a macro:
#   $(name:string1=string2)
#         For each word in 'name' replace 'string1' with 'string2'
# Below we are replacing the suffix .c of all words in the macro SRCS
# with the .o suffix
#
OBJS = $(SRCS:.c=.o)

# define the executable file
MAIN = mycc

#
# The following part of the makefile is generic; it can be used to
# build any executable just by changing the definitions above and by
# deleting dependencies appended to the file from 'make depend'
#

.PHONY: depend clean

all:    $(MAIN)
        @echo  Simple compiler named mycc has been compiled

$(MAIN): $(OBJS)
        $(CC) $(CFLAGS) $(INCLUDES) -o $(MAIN) $(OBJS) $(LFLAGS) $(LIBS)

# this is a suffix replacement rule for building .o's from .c's
# it uses automatic variables $<: the name of the prerequisite of
# the rule(a .c file) and $@: the name of the target of the rule (a .o file)
# (see the gnu make manual section about automatic variables)
.c.o:
        $(CC) $(CFLAGS) $(INCLUDES) -c $<  -o $@

clean:
        $(RM) *.o *~ $(MAIN)

depend: $(SRCS)
        makedepend $(INCLUDES) $^

# DO NOT DELETE THIS LINE -- make depend needs it
```

An example simple Makefile for a Java

```
#
# A simple makefile for compiling three java classes
#

# define a makefile variable for the java compiler
#
JCC = javac

# define a makefile variable for compilation flags
# the -g flag compiles with debugging information
#
JFLAGS = -g

# typing 'make' will invoke the first target entry in the makefile
# (the default one in this case)
#
default: Average.class Convert.class Volume.class

# this target entry builds the Average class
# the Average.class file is dependent on the Average.java file
# and the rule associated with this entry gives the command to create it
#
Average.class: Average.java
        $(JCC) $(JFLAGS) Average.java

Convert.class: Convert.java
        $(JCC) $(JFLAGS) Convert.java

Volume.class: Volume.java
        $(JCC) $(JFLAGS) Volume.java

# To start over from scratch, type 'make clean'.
# Removes all .class files, so that the next make rebuilds them
#
clean:
        $(RM) *.class
```

Here is an example of a Java makefile that uses more advanced makefile syntax.