Lab Report

ECPE 170 – Computer Systems and Networks – Fall 2015

Name: Zhiyun Yan

Lab Topic: C Programming (Language, Toolchain, and Makefiles)

Question(1) Copy and paste in your functional Makefile-1

Answer:

```
all:
	gcc main.c output.c factorial.c -o factorial_program
```

Question(2) Copy and paste in your functional Makefile-2

Answer:

```
all: factorial_program

factorial_program: main.o factorial.o output.o
	gcc main.o factorial.o output.o -o factorial_program

main.o: main.c
	gcc -c main.c

factorial.o: factorial.c
	gcc -c factorial.c

output.o: output.c
	gcc -c output.c

clean:
	rm -rf *.o factorial_program
```

Question(3) Describe - in detail - what happens when the command "make -f Makefile-2" is entered. How does make step through your Makefile to eventually produce the final result?

Answer:
The first time run Make, the it shows
“gcc -c main.c
gcc -c factorial.c
gcc -c output.c
gcc main.o factorial.o output.o -o factorial_program”
This means make command points to “main.c” “factorial.c” and “output.c” file, then gcc compiler them to “factorial_program”. In the second time Make command did nothing because the all file didn't change.

Question (4) Copy and paste in your functional Makefile-3
Answer:

```
all: $(EXECUTABLE)
$(EXECUTABLE): main.o factorial.o output.o
	$(CC) main.o factorial.o output.o -o $(EXECUTABLE)
main.o: main.c
	$(CC) $(CFLAGS) main.c
factorial.o: factorial.c
	$(CC) $(CFLAGS) factorial.c
output.o: output.c
	$(CC) $(CFLAGS) output.c
clean:
	rm -rf *.o $(EXECUTABLE)
```

Question(5) Copy and paste in your functional Makefile-4
Answer:

```
# The variable CC specifies which compiler will be used.
# (because different unix systems may use different compilers)
CC=gcc

# The variable CFLAGS specifies compiler options
#   -c :    Only compile (don't link)
#   -Wall:  Enable all warnings about lazy / dangerous C programming
#  You can add additional options on this same line..
#  WARNING: NEVER REMOVE THE -c FLAG, it is essential to proper operation
CFLAGS=-c -Wall

# All of the .h header files to use as dependencies
HEADERS=functions.h

# All of the object files to produce as intermediary work
OBJECTS=main.o factorial.o output.o

# The final program to build
EXECUTABLE=factorial_program

# -------------------------------------------

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(OBJECTS) -o $(EXECUTABLE)

%.o: %.c $(HEADERS)
	$(CC) $(CFLAGS) -o $@ $<

clean:
	rm -rf *.o $(EXECUTABLE)
```

Question(6) Describe - in detail - what happens when the command "make -f Makefile-4" is entered. How does make step through your Makefile to eventually produce the final result?
Answer:
In Makefile-4, we use variable to let Makefile simple. Makefile-4 can build factorial_program because EXECUTABLE is “factorial_program”. The OBJECT is “main.o factorial.o output.o” and CC is compiler “gcc”. So Makefile-4 can finished work, combine “main.o factorial.o output.o”, and compiler to factorial_program.

Question(7) To use this Makefile in a future programming project (such as Lab 4...), what specific lines would you need to change?
Answer:
The OBJECTS and EXECUTABLE should be changed to other project exactly file name and project name.

Question(8) Take one screen capture of the Bitbucket.org website, clearly showing the "Part 3" source folder that contains all of your Makefiles added to version control, along with the original boilerplate code.

Answer: