CS3844 Lecture 2 Notes:

Objectives:

- Draw the C flow of C compilation from source code to object code.
- Explain the purpose for the preprocessor, compiler, and linker within the C compilation model
- Using the gcc compiler, generate the output for the preprocessor stage of compilation
- Explain the concept of a dependency.
- Create a GNU Make file which automatically generates dependencies, creates preprocessed source code, and links a given C application.

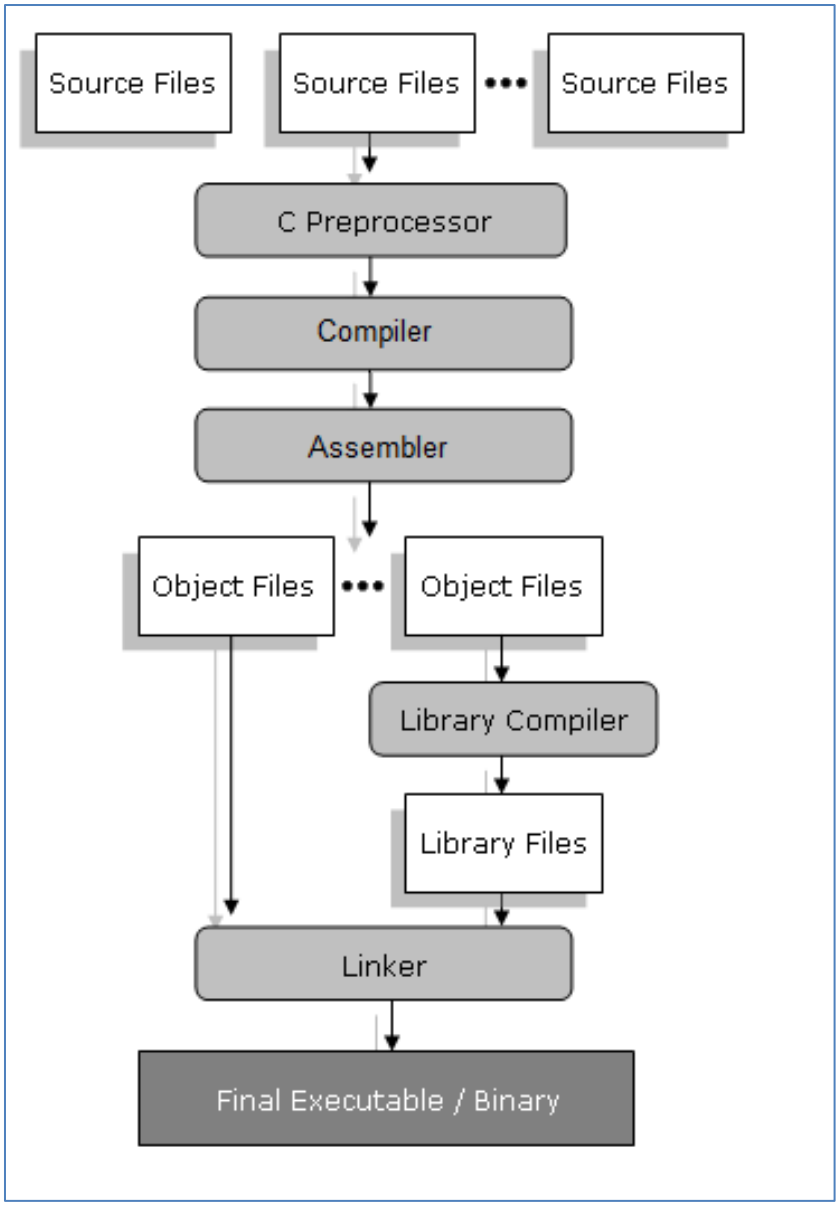
Definitions:

Object file An object file is a module in relocatable format with unexecutable machine code.

Dependency (C definition): A dependency is a file which another file must use in order to compile the compilation phase. For example, the h files which are included in a c file are dependencies of the c file.

Explicit Rule: A specific rule in a makefile which defines how a specific file is to be compiled.

Implicit rule: A general instruction explaining how a given class or set of files are to be translated from one format to another format.



1. C Pre-processor
   1.1. Input: The input to the preprocessor is raw source code
   1.2. Actions taken
      1.2.1.Remove comments from the source code.
      1.2.2.“Copy and paste” include files into the code
      1.2.3.Expand macros and perform appropriate replacements in the source code.
   1.3. Output: preprocessed source code (typically .i file)
   1.4. Generated by giving gcc the –E option
2. Compiler
   2.1. Input: Preprocessed source code

2.2. Actions taken:
    2.2.1. Converts the preprocessed source code into assembly code. Each input file is given its own assembly file.
2.3. Output: Target assembly language code for the given architecture
2.4. Generated by giving gcc the –S option

3. Assembler
    3.1. Input: The assembly language code from the compiler.
    3.2. Actions Taken: Convert the assembly language code into an object file.
    3.3. Output: An object file is generated. The object file is ready to be linked by the linker.
4. Linker
    4.1. Input
        4.1.1. The object files generated by the assembler
        4.1.2. Library files which have been archived
    4.2. Actions Taken
        4.2.1. takes one or more object files generated by a compiler and combines them into a single executable program.
    4.3. Output: Executable program which can be run on the host machine.

Makefiles

A simple makefile consists of "rules" with the following shape:

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as '`clean`' (see Phony Targets).

A *prerequisite* is a file that is used as input to create the target. A target often depends on several files.

A *recipe* is an action that `make` carries out. A recipe may have more than one command, either on the same line or each on its own line. **Please note:** you need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary. If you prefer to prefix your recipes with a character other than tab, you can set the `.RECIPEPREFIX` variable to an alternate character (see Special Variables).

Usually a recipe is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies a recipe for the target need not have prerequisites. For example, the rule containing the delete command associated with the target '`clean`' does not have prerequisites.

A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the recipe on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action. See Writing Rules.

A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this template, but all fit the pattern more or less.

Sample makefile

```
SHELL = /bin/sh

SRCDIR = .

CC = gcc

YACC = bison -y

LIBS =

CFLAGS = -g -save-temps -I. -I$(SRCDIR)

LDFLAGS = -g

# List your sources and objects here.

SOURCES = hello.c

OBJS = hello.o

# list the name of your output program here.

EXECUTABLE = helloWorld


include $(OBJS:.o=.d)


all : $(OBJS) $(EXECUTABLE)


$(EXECUTABLE) : $(OBJS)

	$(CC) $(CFLAGS) $(LDFLAGS) -o $(EXECUTABLE)  $(OBJS) $(LIBS)


%.o : %.c #Defines how to translate a single c file into an object file.

	echo compiling $<

	$(CC) $(CFLAGS) -c $<

	echo done compiling $<
```

%.d : %.c #Defines how to generate the dependencies for the given files. -M gcc option generates dependencies.

```
@set -e; rm -f $@; \

$(CC) $(COMPLIANCE_FLAGS ) -M $< > $@.$$$$; \

sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \

 rm -f $@.$$$$
```

clean : # Clean build in which everything is created from scratch.

```
rm -f *.d

rm -f *.i

rm -f *.o

rm -f *.s

rm -f $(EXECUTABLE)
```