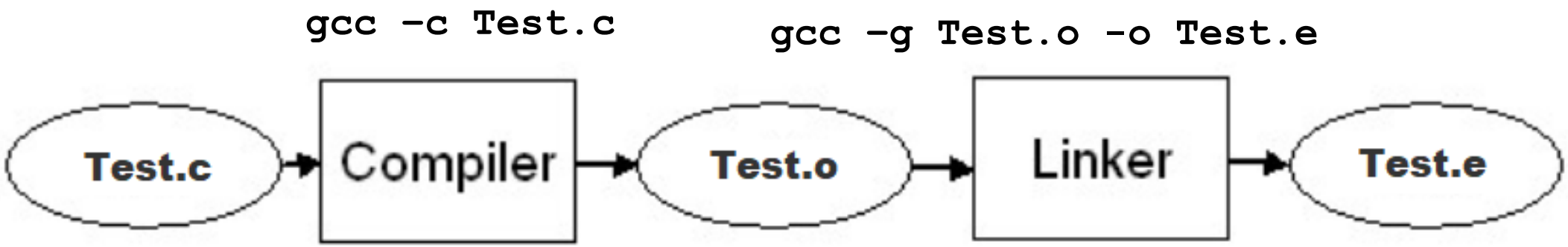
CSE 1320

Week of 02/11/2019

Instructor : Donna French

makefile



The source file that you type into the editor. This is just a text file, anybody can read.

The object file is an intermediate file. It is only readable by the compiler and the linker.

The executable is the final product. It is a binary file that the operating system can run.

```
[frenchdm@omega CA1]$ gcc -c Code1_1000074079.c

[frenchdm@omega CA1]$ ls
Code1_1000074079.c  Code1_1000074079.o

[frenchdm@omega CA1]$ gcc -g Code1_1000074079.o -o
Code1_1000074079.e

[frenchdm@omega CA1]$ ls
Code1_1000074079.c  Code1_1000074079.e  Code1_1000074079.o

[frenchdm@omega CA1]$ Code1_1000074079.e
Decimal to binary convertor

Please enter a decimal number between 0 and 255 170

Decimal 170 converts to binary 10101010
[frenchdm@omega CA1]$
```

makefile

What is a makefile?

make is UNIX utility that is designed to start execution of a makefile.

A makefile is a special file, containing shell commands, that you create and name makefile.

While in the directory containing your makefile, you will type make and the commands in the makefile will be executed.

If you create more than one makefile, be certain you are in the correct directory before typing make.

`makefile`

`make` keeps track of the last time files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the sourcefile up-to-date.

If you have a large program with many source and/or header files, when you change a file on which others depend, you must recompile all the dependent files.

Without a `makefile`, this is an extremely time-consuming task.

`makefile`

As a `makefile` is a list of shell commands, it must be written for the shell which will process the `makefile`. A `makefile` that works well in one shell may not execute properly in another shell.

The `makefile` contains a list of rules. These rules tell the system what commands you want to be executed. Most times, these rules are commands to compile(or recompile) a series of files.

The rules, **which must begin in column 1**, are in two parts. The first line is called a dependency line and the subsequent line(s) are system commands or recipes which must be indented with a tab.

makefile

```
TARGET : DEPENDENCIES
[tab]SYSTEM COMMANDS (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". Multiple target files must be separated by a space

A **dependency** (also called *prerequisite*) is a file that is used as input to create the target. A target often depends on several files. However, the rule that specifies a recipe for the target need not have any dependencies.

The **system command(s)** (also called *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. Recipe lines must be indented using a single <tab> character.

`makefile`

After the `makefile` has been created, a program can be (re)compiled by typing `make` in the correct directory.

`make` then reads the `makefile` and creates a dependency tree and takes whatever action is necessary. It will not necessarily do all the rules in the `makefile` as all dependencies may not need updated. It will rebuild target files if they are missing or older than the dependency files.

Unless directed otherwise, `make` will stop when it encounters an error during the construction process.

makefile

target dependency

```
all : Code1_100074079.e
```

target dependency recipe

```
Code1_100074079.e : Code1_100074079.o
    gcc -g Code1_100074079.o -o Code1_100074079.e
```

target dependency

```
Code1_100074079.o : Code1_100074079.c
    gcc -c Code1_100074079.c
```

recipe

```
[frenchdm@omega CA1]$ more makefile
all : Code1_1000074079.e

Code1_1000074079.e : Code1_1000074079.o
        gcc -g Code1_1000074079.o -o Code1_1000074079.e

Code1_1000074079.o : Code1_1000074079.c
        gcc -c Code1_1000074079.c


[frenchdm@omega CA1]$ make
gcc -c Code1_1000074079.c
gcc -g Code1_1000074079.o -o Code1_1000074079.e

[frenchdm@omega CA1]$ ls
Code1_1000074079.c  Code1_1000074079.e  Code1_1000074079.o
makefile
```

C:\Users\Donna\Desktop\UTA\Coding Assignments\CSE1320 Spring 2019\Coding Assignment 1\makefile - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

makefile

```
all : Code1_1000074079.e

Code1_1000074079.e : Code1_1000074079.o
	gcc -g Code1_1000074079.o -o Code1_1000074079.e

Code1_1000074079.o : Code1_1000074079.c
	gcc -c Code1_1000074079.c
```

```
[frenchdm@omega CA1]$ make
makefile:4: *** missing separator.  Stop.
[frenchdm@omega CA1]$
```

C:\Users\Donna\Desktop\UTA\Coding Assignments\CSE1320 Spring 2019\Coding Assignment 1\makefile - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

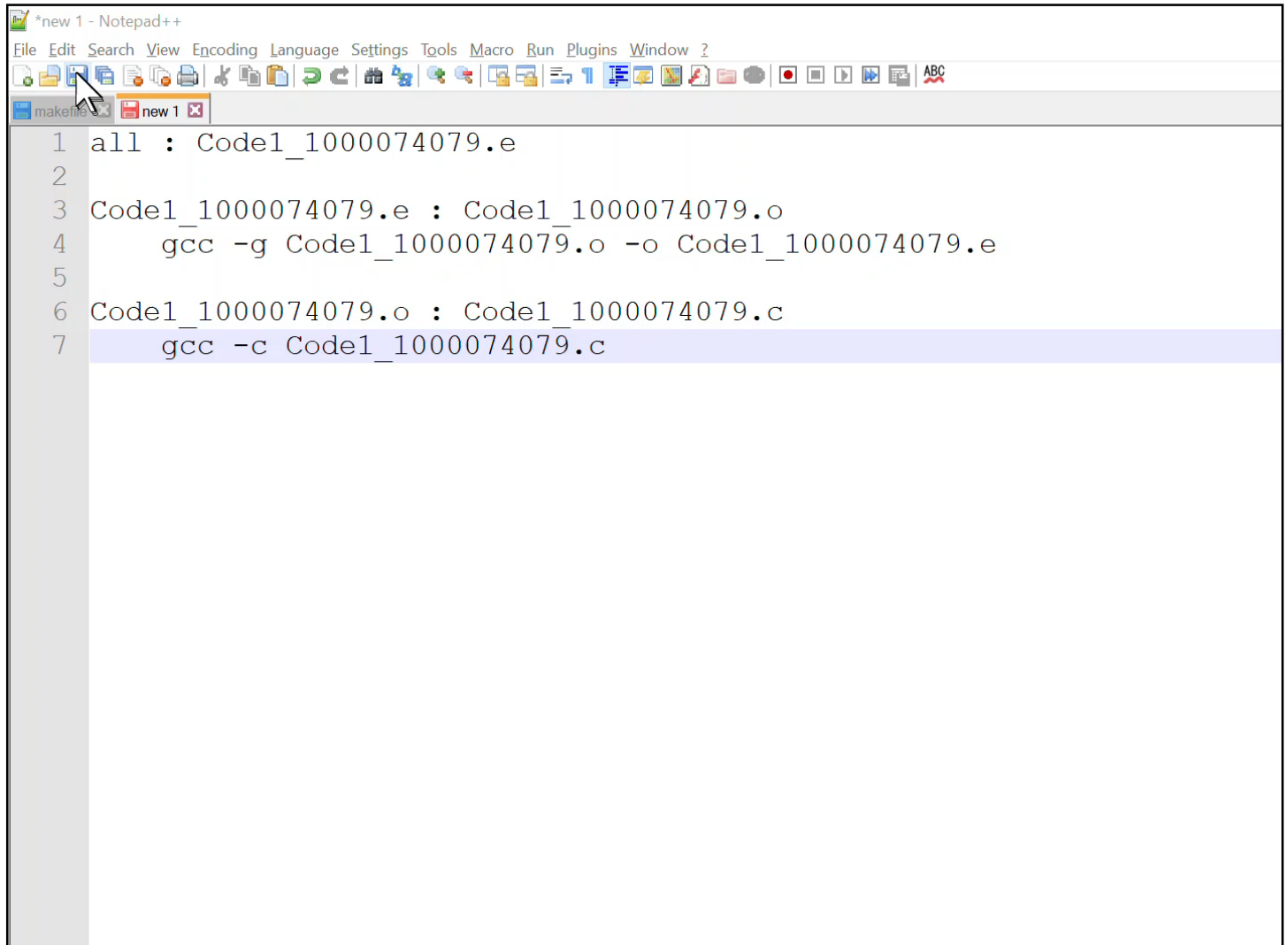
makefile

```
1  all : Code1_1000074079.e CRLF
2  CRLF
3  Code1_1000074079.e : Code1_1000074079.o CRLF
4  	gcc -g Code1_1000074079.o -o Code1_1000074079.e CRLF
5  CRLF
6  Code1_1000074079.o : Code1_1000074079.c CRLF
7  	gcc -c Code1_1000074079.c
```

The name of the `makefile` MUST BE

`makefile`

Save in Notepad++ with a dot on the end to force Notepad++ to not add an extension.



```
[frenchdm@omega CA1]$ ls
Code1_1000074079.c  makefile.txt
[frenchdm@omega CA1]$ make
make: *** No targets specified and no makefile found.  Stop.

[frenchdm@omega CA1]$ mv makefile.txt makefile.mak
[frenchdm@omega CA1]$ ls
Code1_1000074079.c  makefile.mak

[frenchdm@omega CA1]$ make
make: *** No targets specified and no makefile found.  Stop.

[frenchdm@omega CA1]$ mv makefile.mak makefile
[frenchdm@omega CA1]$ make
gcc -c Code1_1000074079.c
gcc -g Code1_1000074079.o -o Code1_1000074079.e
[frenchdm@omega CA1]$
```

makefile

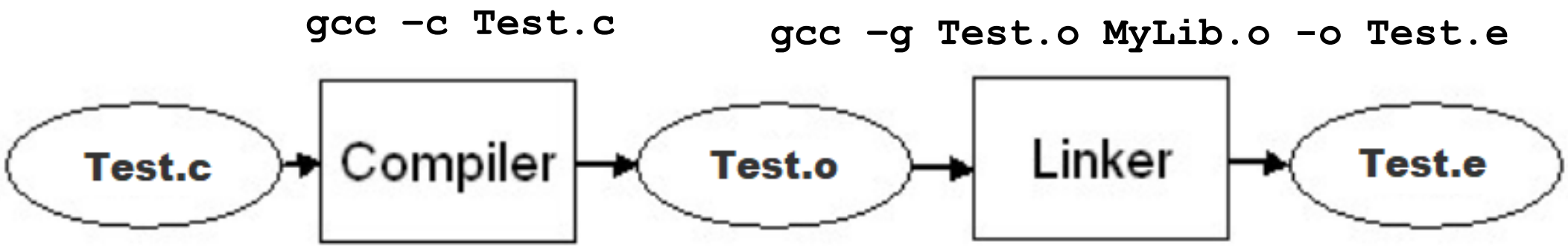
```
SRC = Code2_100074079.c
OBJ = $(SRC:.c=.o)
EXE = $(SRC:.c=.e)

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ)
	gcc $(CFLAGS) $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
	gcc -c $(SRC)
```

```
all : Code1_1000074079.e

Code1_1000074079.e : Code1_1000074079.o
	gcc -g Code1_1000074079.o -o
Code1_1000074079.e

Code1_1000074079.o : Code1_1000074079.c
	gcc -c Code1_1000074079.c
```

makefile

```
SRC = Test.c
OBJ = Test.o
EXE = Test.e

CFLAGS = -g

all : Test.e

Test.e  Test.o

	gcc	-g	Test.o  -o Test.e

Test.o : Test.c
	gcc -c Test.c
```

makefile



The source file that you type into the editor. This is just a text file, anybody can read.

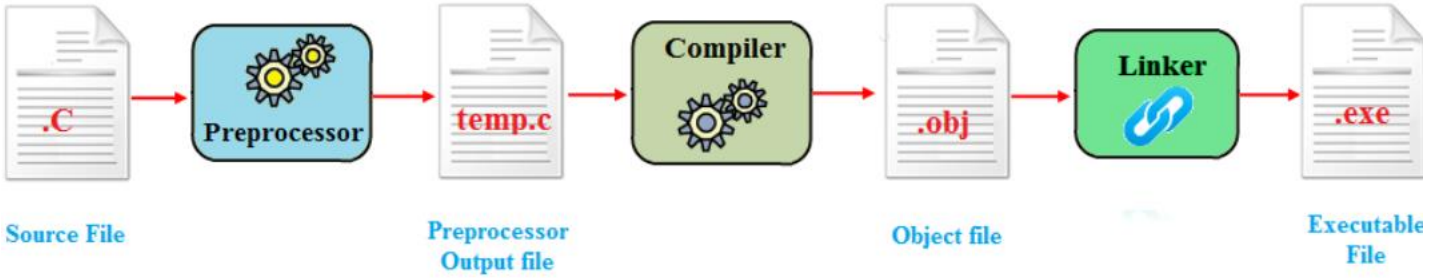
The object file is an intermediate file. It is only readable by the compiler and the linker.

The executable is the final product. It is a binary file that the operating system can run.

makefile

```
SRC1 = Code2_1000074079.c
SRC2 = MyLib.c
OBJ1 = $(SRC1:.c=.o)
OBJ2 = $(SRC2:.c=.o)
EXE = $(SRC1:.c=.e)

HFILES = MyLib.h

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
      gcc $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1) : $(SRC1) $(HFILES)
      gcc -c $(CFLAGS) $(SRC1) -o $(OBJ1)

$(OBJ2) : $(SRC2) $(HFILES)
      gcc -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```

```
SRC = Code2_100074079.c
OBJ = $(SRC:.c=.o)
EXE = $(SRC:.c=.e)

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ)
      gcc $(CFLAGS) $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
      gcc -c $(SRC)
```



compiler

creates an object file

linker

takes in object files and produces an executable file

```
SRC1 = Code2_1000074079.c
SRC2 = MyLib.c
OBJ1 = $(SRC1:.c=.o)
OBJ2 = $(SRC2:.c=.o)
EXE = $(SRC1:.c=.e)

HFILES = MyLib.h

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
        gcc $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1) : $(SRC1) $(HFILES)
        gcc -c $(CFLAGS) $(SRC1) -o $(OBJ1)

$(OBJ2) : $(SRC2) $(HFILES)
        gcc -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```

Library

Libraries are not executable

- do not contain a `main()` function
- only contain functions and declarations

Library

Your library will consist of two files

`MyLib.c`

C file containing your library functions and code

`MyLib.h`

Header file containing the function prototypes for your library

stdio.h

frenchdm@omega:/usr/include

```
[frenchdm@omega include]$ more stdio.h
```

Creating a library

`MyLib.h`

Create `MyLib.h` and move prototypes from `Code2.c` to `MyLib.h`

Add `include` guard

`MyLib.c`

Create `MyLib.c` and move `ConvertDecimalToBinary()` and `PrintBinary()` code from `Code2.c` to `MyLib.c`

Add includes

```
stdio.h
MyLib.h
```

`Code2.c`

Add include for `MyLib.h`

`makefile`

Create a `makefile` that compiles/links two object files.

Special Note about Your Header Files

If you look at any system include you will see

```
frenchdm@omega:~
#ifndef _STDIO_H

#if !defined __need_FILE && !defined __need___FILE
# define _STDIO_H      1
```

at the beginning and

```
#endif /* !_STDIO_H */
```

at the end

Special Note about Your Header Files

In the **C** and **C++** programming languages, an #include **guard**, sometimes called a macro **guard** or **header guard**, is a particular construct used to avoid the problem of double inclusion when dealing with the include directive. The addition of #include **guards** to a **header** file is one way to make that file idempotent.

Idempotence is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application.

Special Note about Your Header Files

Add `#include` **guard to** `MyLib.h`

```
#ifndef _MYLIB_H
#define _MYLIB_H

void ConvertDecimalToBinary(int, int []);
void PrintBinary(int []);

#endif
```

Compiling and Linking

source files for one executable

Module A	**Module B**	**Module C**
Contains main()	Functions to open files	Functions to perform FTP actions
Prompts user for input	Functions to read files	
Call various functions based on input	Functions to write to files	

an object file is created for each module and then the linker puts the objects together to create an executable

```
SRC1 = Code2_1000074079.c
```

Module A – Code2_1000074079.c

```
SRC2 = MyLib.c
```

Module B – MyLib.c

```
OBJ1 = $(SRC1:.c=.o)

OBJ2 = $(SRC2:.c=.o)

EXE = $(SRC1:.c=.e)

HFILES = MyLib.h

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
        gcc $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)
```

Link both object files together to make an executable

```
$(OBJ1) : $(SRC1) $(HFILES)
        gcc -c $(CFLAGS) $(SRC1) -o $(OBJ1)
```

Generate object file for Code2_1000074079.c

```
$(OBJ2) : $(SRC2) $(HFILES)
        gcc -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```

Generate object file for MyLib.c