

LAB 5: COMPILATION STEPS OF A C PROGRAM & MAKE FILE

Dr. Kazi Md. Rokibul Alam & JAKARIA

COMPILATION STEPS OF A C PROGRAM

- To compile and run the C program **helloworld.c**, all C statements must be translated individually into a sequence of instructions that a machine can understand.
- These instructions are then packaged in a form called executable object program. There are other programs which perform this task to get the program running.
- On a UNIX/Linux system, the translation from source code to object code (executable) is performed by a compiler driver. Here we will compile C program by gcc.

```
#include <stdio.h>
int main()
{
    printf("hello, world!\n");
    return 0;
/* helloworld.c */
}
```

COMPILATION STEPS OF A C PROGRAM

- The following command (provided that **gcc** is installed on your Linux box) compiles C program **helloworld.c** and creates an executable file called **helloworld**. Don't forget to set appropriate permissions to **helloworld.c**, so that you won't get execute permission errors.
- **[root@host ~]# gcc helloworld.c -o helloworld**
- While compiling **helloworld.c** the gcc compiler reads the source file **helloworld.c** and translates it into an executable **helloworld**. The compilation is performed in four sequential phases by the compilation system (a collection of four programs - **preprocessor, compiler, assembler, and linker**).

COMPILATION STEPS OF A C PROGRAM

- Now, let's perform all four steps one by one and understand independently.
- **1. Preprocessing**
- During compilation of a C program the compilation is started off with preprocessing the directives (e.g., `#include` and `#define`).
- The preprocessor (`cpp --c preprocessor`) is a separate program in reality, but it is invoked automatically by the compiler. For example, the `#include <stdio.h>` command in line 1 of `helloworld.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another file typically with the `.i` suffix. In practice, the preprocessed file is not saved to disk unless the `-save-temps` option is used.

COMPILATION STEPS OF A C PROGRAM

- This is the first stage of compilation process where preprocessor directives (macros and header files are most common) are expanded. To perform this step gcc executes the following command internally.
- [root@host ~]# `cpp helloworld.c > helloworld.i`
- [root@host ~]# `gcc -E helloworld.c -o helloworld.i`
- The result is a file **helloworld.i** that contains the source code with all macros expanded. If you execute the above command in isolation then the file **helloworld.i** will be saved to disk and you can see its content by vi or any other editor you have on your Linux box.

(or) This are the commands

COMPILATION STEPS OF A C PROGRAM

- **2. Compilation**
- In this phase compilation proper takes place. The compiler (cc) translates **helloworld.i** into **helloworld.s**. File **helloworld.s** contains assembly code.
- You can explicitly tell gcc to translate **helloworld.i** to **helloworld.s** by executing the following command.
- **[root@host ~]# gcc -S helloworld.i -o helloworld.s**
- The command line option **-S** tells the compiler to convert the preprocessed code to assembly language without creating an object file. After having created **helloworld.s** you can see the content of this file.

COMPILATION STEPS OF A C PROGRAM

- **3. Assembly**
- Here, the assembler (as) translates **helloworld.s** into machine language instructions, and generates an object file **helloworld.o**. You can invoke the assembler at your own by executing the following command.
- **[root@host ~]# as helloworld.s -o helloworld.o**
- The above command will generate **helloworld.o** as it is specified with **-o** option. And, the resulting file contains the machine instructions for the classic "Hello World!" program, with an undefined reference to printf.

↑
printf - its definition
system provide `printf.o`

COMPILATION STEPS OF A C PROGRAM

- **4. Linking**
- This is the final stage in compilation of "Hello World!" program. This phase links object files to produce final executable file. An executable file requires many external resources (system functions, C run-time libraries etc.).
- Regarding our "Hello World!" program you have noticed that it calls the printf function to print the 'Hello World!' message on console. This function is contained in a separate pre compiled object file **printf.o**, which must somehow be merged with our **helloworld.o** file.
- The linker (ld) performs this task for you. Eventually, the resulting file **helloworld** is produced, which is an executable. This is now ready to be loaded into memory and executed by the system.
- [root@host ~]# **gcc helloworld.o -o helloworld**
- [root@host ~]# **gcc helloworld1.o helloworld2.o -o helloworld** [for multiple object file]

MAKE FILE

ପ୍ରାୟତଃ ସବୁ କିଛି
ସାଧାରଣତଃ ରିକମ୍ପାଇଲ
କରିବା ଆବଶ୍ୟକ

- Small C/C++ applications with a couple of modules are easy to manage. Developers can recompile them easily by calling the compiler directly, passing source files as arguments. That is a simple approach. However, *Particularly* when a project gets too complex with many source files it becomes necessary to have a tool that allows the developer to manage the project.
- The tool we are talking about is the **make** command. The **make** command is used not only to help a developer compile applications, it can be used whenever you want to produce output files from several input files.
- This tutorial focuses on C applications and how to use the **make** command and **makefile** to build them. There is a **sample** folder under **make_samples**. The most important files in the samples are the **makefiles** not the C source code.

MAKE FILE

- Make Tool: **Syntax Overview**
- **make** command syntax is:
- **make [options] [target]**
- You can type **make --help** to see all options **make** command supports. In this tutorial an explanation of all those options are not in the scope. The main point is **makefile** structure and how it works. **target** is a tag (or name defined) present in **makefile**.
- **make** requires a **makefile** that tells it how your application should be built. The **makefile** often resides in the same directory as other source files and it can have any name you want. For instance, if your **makefile** is called **run.mk** then to execute **make** command type:
make -f run.mk

MAKE FILE

- **-f** option tells **make** command the **makefile** name that should be processed.
- There are also two special names that makes **-f** option not necessary:
makefile and **Makefile**. If you run **make** not passing a file name it will look first for a file called **makefile**. If that does not exist it will look for a file called **Makefile**. If you have two files in your directory one called **makefile** and other called **Makefile** and type:

make <enter>

make command will process the file called **makefile**. In that case, you should use **-f** option if you want **make** command processes **Makefile**.



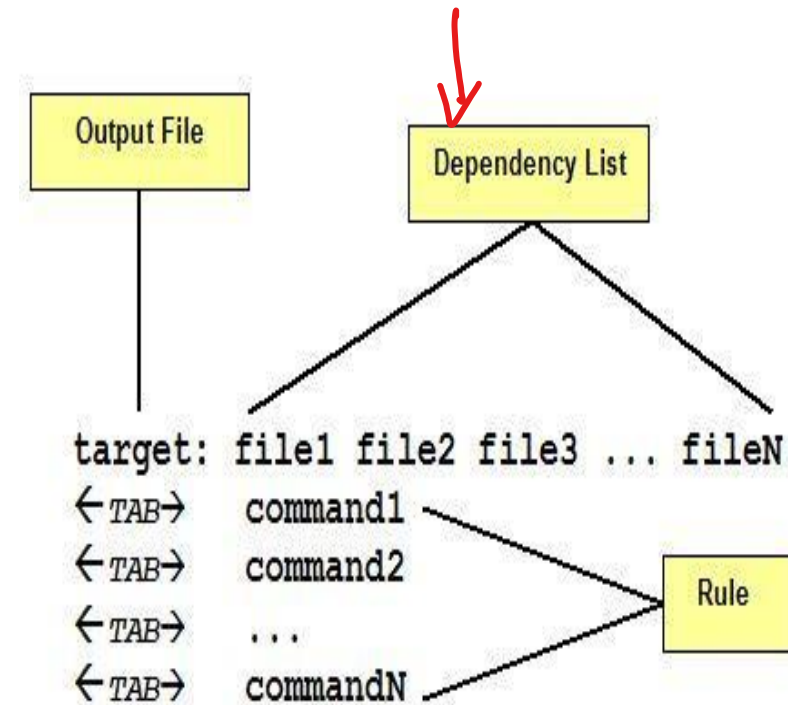
MAKE FILE

A make file consists of a set of **targets**, **dependencies** and **rules**.

A **target** most of time is a file to be created/updated. **target** depends upon a set of source files or even others **targets** described in **Dependency List**. **Rules** are the necessary commands to create the **target** file by using **Dependency List**.

As you see in **figure** each command in the **Rules** part must be on lines that start with a **TAB** character. Space issue errors. Also, a space at end of the **rule** line may cause **make** issues an error message.

The **makefile** is read by **make** command which determines **target** files to be built by comparing the dates and times (timestamp) of source files in **Dependency List**. If any dependency has a changed timestamp since the last build **make** command will execute the rule associated with the **target**.

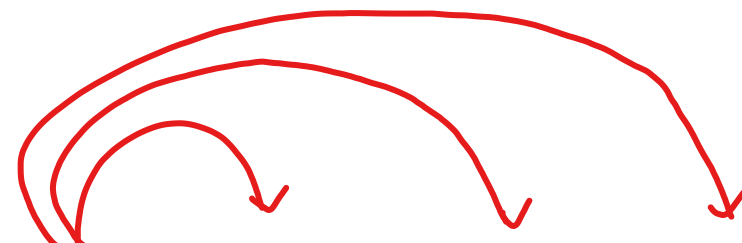


MAKE FILE

- **sample** is an example of a simple **makefile**, there are multiple **targets**. There are 2 **makefiles**: *mkfile.r* and *mkfile.w* to demonstrate the right and the wrong way to write a **makefile**.
- As you notice, the final executable (**app target**) is formed by 3 object files: *main.o*, *mod_a.o* and *mod_b.o*. Each one is a **target** with its source files that represent its **dependency list**.
- **app target** is the main **target** or the **target** that will result in the main executable file. Notice **app dependency list**. They are names of other **targets**.
- Both **makefiles** are complete. The main difference is the order the **targets** are placed in the **makefile**.
- So, we have:

mkfile.r and *mkfile.w*

MAKE FILE



```
app: main.o mod_a.o mod_b.o
    cc -o app main.o mod_a.o mod_b.o
main.o: main.c inc_a.h inc_b.h
    cc -c main.c

mod_a.o: mod_a.c inc_a.h
    cc -c mod_a.c

mod_b.o: mod_b.c inc_b.h
    cc -c mod_b.c
```

output name

mkfile.r

MAKE FILE

```
main.o: main.c inc_a.h inc_b.h  
    cc -c main.c
```

```
mod_a.o: mod_a.c inc_a.h  
    cc -c mod_a.c
```

```
mod_b.o: mod_b.c inc_b.h  
    cc -c mod_b.c
```

```
app: main.o mod_a.o mod_b.o  
    cc -o app main.o mod_a.o mod_b.o
```

mkfile.w

MAKE FILE

- Let us try the following sequence of commands:

```
[root@localhost sample2]#  
[root@localhost sample2]# make -f mkfile.w _____ 1  
cc -c main.c  
[root@localhost sample2]# rm -f *.o _____ 2  
[root@localhost sample2]# make -f mkfile.r _____ 3  
cc -c main.c  
cc -c mod_a.c  
cc -c mod_b.c  
cc -o app main.o mod_a.o mod_b.o  
[root@localhost sample2]# ./app _____ 4  
  
Hello, I m func_a!  
  
Hello, I m func_b!  
[root@localhost sample2]# rm -f *.o app _____ 5  
[root@localhost sample2]# make -f mkfile.w app _____ 6  
cc -c main.c  
cc -c mod_a.c  
cc -c mod_b.c  
cc -o app main.o mod_a.o mod_b.o  
[root@localhost sample2]# ./app  
  
Hello, I m func_a!  
  
Hello, I m func_b!  
[root@localhost sample2]#
```


MAKE FILE

- **make** command is invoked to process *mkfile.w* and you can see only the first rule is executed.
- All object files resulted from previous builds were removed to force **make** command to perform a full build.
- **make** command is invoked to process *mkfile.r* and all modules are correctly created.
- **app** is executed.
- All objects and executables were removed to force the **make** command to perform a full build.
- **make** command is invoked to process *mkfile.w* again. But this time **app target** is passed as an argument and all modules are correctly created.

MAKE FILE

- So, what is wrong with *mkfile.w* ? Well, technically nothing when you inform the **main target** (**figure 3 - item 6**). However, when you do not inform a **target** the **make** command reads **makefile** from the beginning to find the first **target** to process. In the *mkfile.w* case, that **target** is *main.o*. **main.o target** only says to **make** to build *main.o* from *main.c*, *inc_a.h* and *inc_b.h* - there is nothing more related to do. Make will not read the next **target**.
- **Note:** the **first target** read determines how make must interpret all other **targets** and which order it must follow during the building process. So, the **first target** should be the **main target** and it might relate to one or more secondary **targets** to perform the build.

MAKE FILE

- Let us see **app target**. It is placed in different lines in both **makefiles** but they have identical syntax in both. So, item **3** and **item 6** of **figure 3** will produce the same result:
- **app target** says to **make** command it has 3 dependency to process first: *main.o*, *mod_a.o* and *mod_b.o* before building the final executable (**app**).
- Then, **make** starts finding for a *main.o* target and process it.
- After, it finds and processes *mod_a.o*.
- And finally, *mod_b.o* is processed.
- When all those 3 **targets** are built, **app target rule** is processed and **app** executable is created.



MAKE FILE

- Sometimes a **target** does not mean a **file** but it might represent an action to be performed. When a **target** is not related to a file it is called **phony target**.
- For instance:
 getobj:
 mv obj/*.o . 2>/dev/null
- **getobj target** move all files with **.o** extension from *obj* directory to current directory -- not a big deal. However, you should be asking yourself: "What if there is no file in **obj** ?" That is a good question. In that case, the **mv** command would return an error that would be passed to the **make** command.



MAKE FILE

- **Note:** **make** command default behavior is to abort the processing when an error is detected while executing commands in **rules**.
- Of course, there will be situations that the *obj* directory will be empty. How will you avoid the **make** command from aborting when an error happens?
- You can use a special character - (minus) preceding the **mv** command.
Thus:

getobj:

```
-mv obj/*.o . 2>/dev/null
```



MAKE FILE

- There is a special **phony target** called **all** where you can group several **main targets** and **phony targets**. **all phony target** is often used to lead **make** command while reading **makefile**.
- For instance:

all: getobj app install putobj



MAKE FILE

- The **make** command will execute the **targets** in sequence: **getobj**, **app**, **install** and **putobj**.
- Another interesting feature, **make** command supports is the concept of **MACRO** in **makefiles**. We can define a MACRO by writing:

MACRONAME=value

- And access the value of MACRONAME by writing either $\$(\text{MACRONAME})$ or $\${\text{MACRONAME}}$.





MAKE FILE

- For instance:

EXECPATH=./bin

INCPATH=./include

OBJPATH=./obj

CC=cc

CFLAGS=-g -Wall -I\$(INCPATH)

- While executing, **make** replaces \$(MACRONAME) with the appropriated definition. Now we know what **phony targets** and **macros** are.

Attention Please 😊

- You must practice these workflow using given **files** for proper understanding.
- Otherwise you can not answer correctly to your VIVA questions which will be held after next LAB.
- Thank you . 😊