CPE-2600 FALL 2024

Systems Programming

Lab 3 Makefiles

Objectives

- Observe the action of 'git clone'
- Explain the purpose for a makefile
- Define the terms target and dependency
- Compare and contrast implicit and explicit rules
- Construct a simple makefile

Preparation

This exercise will be using a 3rd party library to control the console known as "ncurses." Ncurses stands for "New Curses" as it is a reimplementation of a much older library, Curses.

Your system would likely support an existing program that uses ncurses because the dynamic libraries for ncurses are installed by default. However, in order to build a program from source that uses the library you will need to install the "development package" that consists of header files and static libraries.

So, do this first:

```
sudo apt update

sudo apt install libncurses-dev
```

GIT Intro and git clone

git is a distributed version control system. We will increasingly make use of throughout the course. As an introduction we will start with a single command, 'git clone'. This command will essentially copy or clone a remote repository to the local system (in our case). Once cloned we can work on the repository independently. Later, the repository can be pushed back to the remote repository, but that is a topic for a later exercise.

The code for this exercise is located in a repository at https://github.com/MSOE-Rothe/snake.git. To create a clone, simply issue the command 'git clone https://github.com/MSOE-Rothe/snake.git' from your WSL command line. Observe that this command will have created a new directory 'snake' and placed source files in that directory.

Take a quick look at the files – there are a handful of source and header files which is typical for a program such as this.

Try to build this program. It obviously has multiple source files (HINT) and needs to link with a non-standard library (HINT). The name of the external library is ncurses (HINT).

Do not move on until you are able to build and run the program.

SUBMISION REQUIREMENT - include a screenshot of the successful build (showing command used to build on the console).

Make and Makefiles

You may have noticed that the command to get gcc to compile your project can get lengthy. You have to list all of the source files, linker options, warning and error options, etc. This is hard to keep track of and can result in inconsistent handling of builds. One solution would be to put the exact command into a shell script, and there is nothing wrong with that approach. However, every time you invoke gcc in this fashion it will build the entire project whether it needs it or not. Imagine a large project with perhaps hundreds of source files. You make one minor change and you may be faced with a lengthy rebuild.

Make is a tool that can help manage projects for us, and its biggest benefit is that is allows incremental building. So, when a change to one source file is made, only that source file is recompiled. All of the other source files are kept in an intermediate form where they are already compiled but not yet linked (aka object files). Then, once the one changed source file is compiled the project can be relinked to produce an executable.

Of course, this feature comes at a price. A new "language" if you will. Using the "language" of make, you create a "Makefile." Then, to build your project, you simply issue the command "make" and the make tool will find the makefile, and incrementally build your project.

In its very basic form, a makefile consists of one or more blocks in the form:

target: dependencies

[tab] system command

The first line is referred to as the dependency line and the second line, which is tabbed in from the dependency line is the command line.

target is a reference to what is being created and can be the name of something make will produce, or, could be a pattern match to intermediate ingredients to the final build. The name of a target can also be passed into make as an argument – if there is no argument, the first target in the makefile will be produced.

dependencies are the ingredients necessary to build the target. Make will ensure the dependencies are current (have not changed since the last build). If a dependency has changed, make may have to rebuild it using the information in another block within the makefile.

[tab] is a literal tab character that make uses to associate related lines.

Finally, system command is the command make will invoke to build the target.

There are typically several blocks for a C project and as stated. Some of the blocks are considered explicit if it is referring to a specific target like the executable, and other are implicit which uses pattern matching based on file extensions.

In addition to the build rules, makefiles typically contain macros/symbols/variables to be able to easily change command for building certain targets. For example, the symbol CFLAGS is usually used to pass compiler flags to the compile command. By defining this symbol at the top of the makefile, it can easily be edited to change build options.

Dependencies

Make relies on knowing dependencies. Each source file is compiled independently, yet, relies on certain information present in other files. The other files are brought in through the #include mechanism. Often one header file may include another and so on. Can we get a picture of this? Yet another command line option for gcc allows just that. The -M option will ask gcc to only display (to stdout) the dependencies for a given source file. Try this at the command line: gcc snake.c -M. Study the output generated and compare to the #include statements at the top of snake.c. Only two files were included, but gcc lists about 30 header files as dependencies. Why?

SUBMISION REQUIREMENT - explain why so many header files are listed by gcc when only two files are included. Include at least one screen cap as proof.

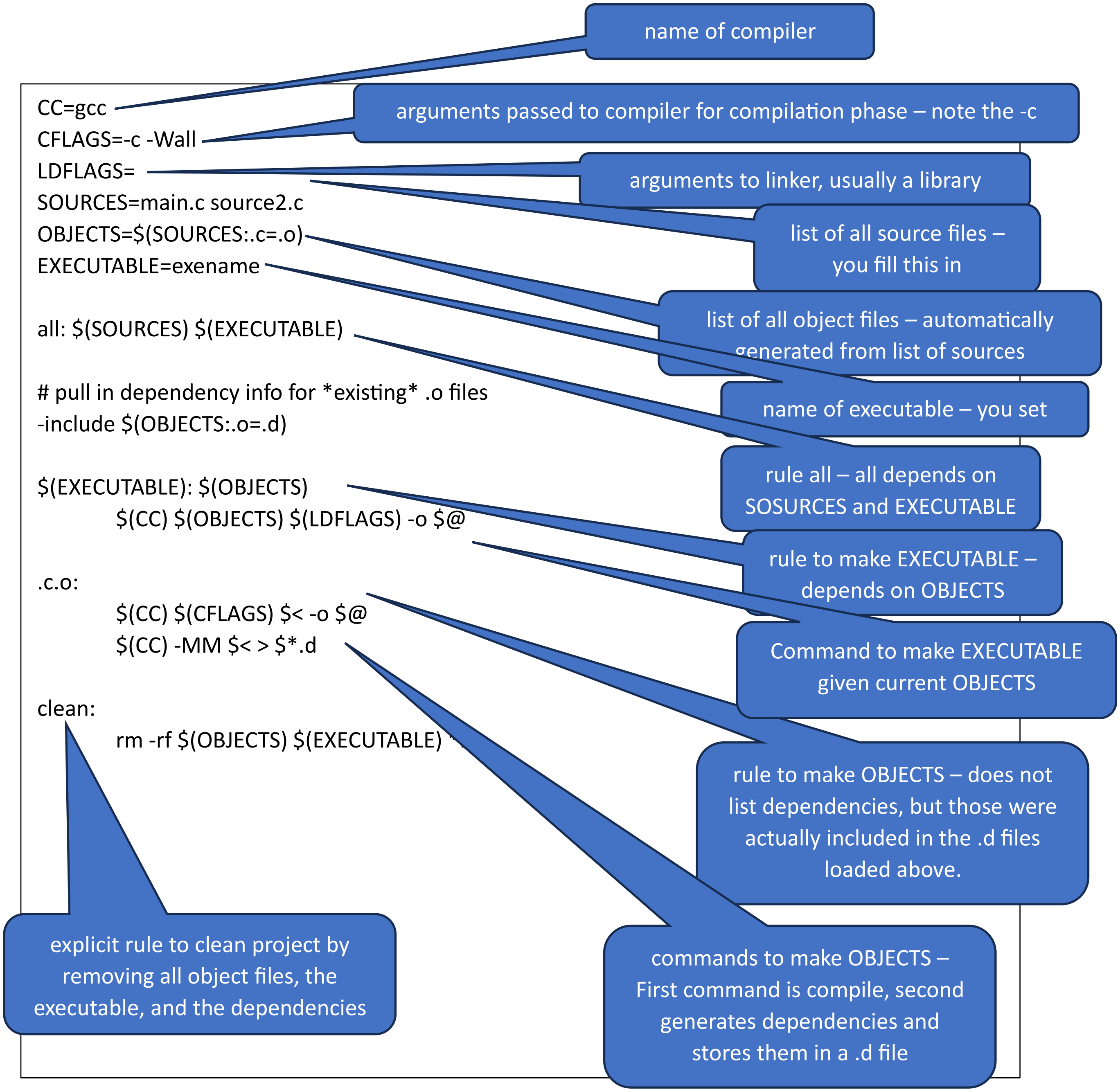
While we are interested in all of these dependencies, what we are really interested in are the dependencies that are likely to change during the course of our development. The standard library includes are not likely to change, so it might be nice to view just the “local” dependencies. -MM will do just that. Run the command: gcc snake.c -MM. Notice the much more relevant list. Our makefile will need this information so that it can automatically recompile snake.c iff snake.c has change or if any of the local dependencies have changed.

We could copy and paste the results for gcc -MM into our makefile for each source file, but that would not be very robust. Rather, it would be nice if the dependency information could be built automatically. The makefile below does just that. See if you can figure out approximately how this works.

A Simple Makefile

Perhaps you now have the sense that makefiles can get complicated. That is very true. However, we can try to keep them simple too. The following represents a relatively simple makefile that will allow us to take advantage of the incremental build capability of make. Please pay close attention to the description of each line. Some of the syntax may not make sense, but the overall purpose should be understandable.

The example here is not customized for our current project. That is something you must do.



```
CC=gcc
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.c source2.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=exename

all: $(SOURCES) $(EXECUTABLE)

# pull in dependency info for *existing* .o files
-include $(OBJECTS:.o=.d)

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

.c.o:
	$(CC) $(CFLAGS) $< -o $@
	$(CC) -MM $< > $*.d

clean:
	rm -rf $(OBJECTS) $(EXECUTABLE)
```

name of compiler

arguments passed to compiler for compilation phase – note the -c

arguments to linker, usually a library

list of all source files – you fill this in

list of all object files – automatically generated from list of sources

name of executable – you set

rule all – all depends on SOSURCES and EXECUTABLE

rule to make EXECUTABLE – depends on OBJECTS

Command to make EXECUTABLE given current OBJECTS

rule to make OBJECTS – does not list dependencies, but those were actually included in the .d files loaded above.

explicit rule to clean project by removing all object files, the executable, and the dependencies

commands to make OBJECTS – First command is compile, second generates dependencies and stores them in a .d file

Using this makefile as a starting point, create a makefile for the snake project. Hint – all you should need to do is edit LDFLAGS, SOURCES, and EXECUTABLE. Save your makefile with the name "Makefile" in the same directory as the .c and .h files of your project.

To test your makefile - start by listing the contents of your working space. There really should not be anything other than source files and header files. Then issue the command 'make'. Since all is the first explicit rule, this will be the same as 'make all'. Observe the commands echoed to the console.

SUBMISION REQUIREMENT - grab a screen cap of the console with your first successful build.

List the files in your project directory. Note there is now an object file (.o) for each source file and a dependency file (.d) for each source. We know what the .o files are from the previous two labs. Take a look at the .d files.

Run the program, but do not get too distracted by this highly engaging game.

Testing Incremental Build

SUBMISION REQUIREMENT - answer all of these questions in your submission. Use screenshots as desired to communicate your answer.

Without changing any of the source files, type the command 'make' again. What happens? Does this make sense?

Delete field.o (rm field.o). Run 'make'. What happens? Does this make sense?

Edit field.c (or just update its timestamp with 'touch field.c') and Run 'make'. What happens? Does this make sense? How did make know to recompile field.c?

Edit field.h (or just update its timestamp with 'touch field.h') and Run 'make'. What happens? Does this make sense? Be very specific with this answer.

Issue command 'make clean'. What happens?

Deliverable

Collect various screen caps and answers to questions and submit a pdf to the appropriate Canvas assignment.

Also, from now on, you must use a makefile for all of your programming assignments. The template here is highly recommended and will work well for most uses.

A Final Thought

After the build is complete, you will notice the project directory is pretty cluttered. You have the original source code and header files, of course. But you also now have an object (.o) and a dependency (.d) for each source file. For the incremental build to work, you do have to keep these files. In a future improvement we will modify our makefile to store them in a subdirectory so as to not clutter our project directory.

Employment opportunities in dev ops is extremely valuable to an organization. One of the many duties of a dev ops person is to manage the build server and git repos and security concerns.