HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 2
C Programming on Linux
Course: CO2018 Operating Systems - Fall 2018

Phuong-Duy Nguyen, Duc Hai Nguyen, Truc D. T. Nguyen, Hoang-Anh Pham

August 31, 2018

Goals:

- Review basic shell commands on Linux, Mac OS X.
- Understand how to compile and run programs on Linux, Mac OS X.
- Review C programming language

Contents:

- Programming with C language.
- Compile a program with Makefile.

Score:

- 30% in-class performance
- 70% e-learning submission

1. Introduction

1.1. C programming on Linux/Mac OS X

GNU C Coding Standards

- **Keep the length of source lines to 79 characters or less, for maximum readability in the widest range of environments.**
- Put a comment on each function saying what the function does, what sorts of arguments it gets, and what the possible values of arguments mean and are used for.
- Please explicitly declare the types of all objects. For example, you should explicitly declare all arguments to functions, and you should declare functions to return int rather than omitting the int.

Reference: `http://www.gnu.org/prep/standards/standards.html`. Formatting your source code

Compiling process: While some computer languages (e.g., Scheme or Basic) are normally used with interactive interpreters (where you type commands to execute immediately), C source codes are always compiled into binary code by a program called "compiler." And then it can be executed by users. This is actually a multi-step process which is illustrated in the following figure.

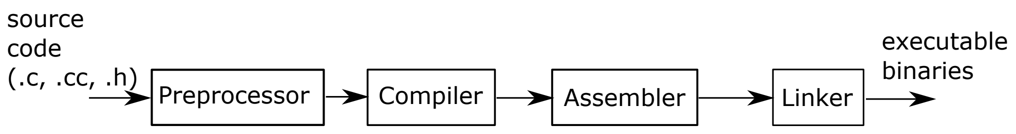

Figure 1.1: C Compiling scheme

Steps in compiling process:

- Preprocessor
- Compiler
- Assembler
- Linker

Compilers and Libraries: Apple provides a customized/optimized GNU CC, with backends for C, C++, Objective-C and Objective-C++. Compilers for many other languages are available either precompiled (such as the XL Fortran Advanced Compiler from IBM), or can be compiled from source, which is not any harder in general than compiling the same source on, say, Linux or FreeBSD. The LLVM compiler is the next-generation compiler, introduced in Mac OS X. In Xcode of Mac OS X, the LLVM compiler uses the Clang front end to parse source code and turn it into an interim format.

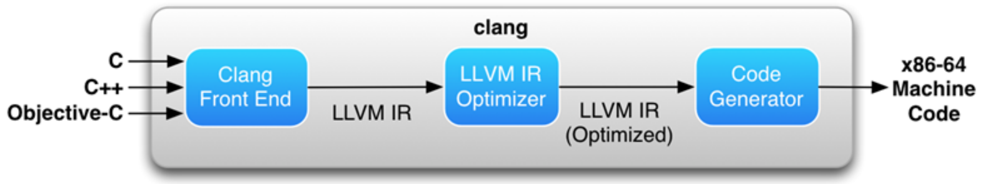

Figure 1.2: Clang in Mac OS X

Figure below shows a C program compiled in step by step.

```
% Preprocessed source file
$ gcc -E [-o hello.cpp] hello.c

% Assembly code
$ gcc -S [-o hello.S] hello.c

% Binary file
$ gcc -c [-o hello.o] hello.c

% Executable file
$ gcc [-o hello] hello.c
```

2. Practice

2.1. Compile and run a program

Steps for developing a program

In general, the compiling progress includes these steps:

1. Create source code file hello.c

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  printf("Hello, World!\n");
  return 0;
}
```

2. Generate object file(s):

```
$ gcc -c souce_code_file.c
# Example:
$ gcc -c hello.c
# or
$ gcc -c -o hello.o hello.c
```

3. Generate an executable file:

```
$ gcc -o executable_file object1.o object2.o ...
# Example:
$ gcc -o hello hello.o
```

We can compile the program directly from the source code file without the step of generating the object file. However, it could be more challenging in identifying errors.

4. Generate an executable file:

```
$ gcc -o executable_file src1.c src2.c ...
# Example:
$ gcc -o hello hello.c
```

5. Run the program:

```
$ ./executable_file
# Example: to list the crated executable binary file
$ ls
```

```
hello    hello.c          hello.o
# To execute the binary file
$ ./hello
```

- During compiling a program, the source code can make some errors. The compiler provides debuggers used to show the information of each error. The structure of showing errors: `<file>:<row>:<column_letter>:<type>:<detail>`
- For example, error 1:

```
$ gcc -o hello.o -c hello.c
hello.c:1:18: fatal error: stdo.h: No such file ...
compilation terminated.
```

- From the example of error 1:
  - Error file: hello.c
  - Error line: 1
  - The column of error letter: 18
  - Type of error: error
  - Detail info: stdo.h not found

2.2. Makefile

A makefile is a file containing a set of directives used with the **make** build automation tool. Most often, the makefile directs **make** on how to compile and link a program. Using C/C++ as an example, when a C/C++ source file is changed, it must be recompiled. If a header file has changed, each C/C++ source file that includes the header file must be recompiled to be safe. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable program.[1] These instructions with their dependencies are specified in a makefile. If none of the files that are prerequisites have been changed since the last time the program was compiled, no actions take place. For large software projects, using Makefiles can substantially reduce build times if only a few source files have changed. A makefile consists of "rules" in the following form:

```
# comment
# (note: the <tab> in the command line
# is necessary for make to work)

target: dependency1 dependency2 ...
        <tab> command
```

Where,

- target: 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".
- dependency1, dependency2,...: 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 prerequisites. For example, the rule containing the delete command associated with the target "clean" does not have prerequisites.
- command: Needed commands is used for performing rules.

For example, we have three source code files including main.c, hello.h, hello.c.

```
// File: main.c
#include "hello.h"

int main() {
        helloworld();
        return 0;
}
```

```
// File: hello.h

void helloworld(void);
```

```
// File: hello.c
#include "hello.h"
#include <stdio.h>

void helloworld(void) {
        printf("Hello, world\n");
}
```

In this example, we compile .c files into object files .o, and then link all of object files into a single binary. Firstly, that is the process of compiling source code files into object files.

- main.o: main function in main.c calls helloworld() which is declared in hello.h. Thereby, to compile main.c, we need the information declared from hello.h. To create main.o, we need hello.h and main.c. Therefore, the rule for creating main.o is:

```
main.o: main.c hello.h
        gcc -c main.c
```

- hello.o: similar to the rule of main.o, we need two files named hello.c and hello.h to create hello.o. Note that hello.c using printf() in the library stdio.h to print the output on screen. However, this is the library integrated with GCC, so we do not need to fill in the dependency of the rule.

```
hello.o: hello.c hello.h
        gcc -c hello.o
```

- hello: Because helloworld is declared in hello.h, but it is defined in hello.c and compiled into the binary in hello.o, therefore, if the main function calls this function, we need to link hello.o with main.o to create the final binary. This file depends on hello.o and main.o.

```
all: main.o hello.o
        gcc main.o hell.o -o hello
```

- Finally, we can add the rule of clean to remove all of object files and binaries in case of compiling an entire program.

```
clean:
        rm -f *.o hello
```

The final result of Makefile:

```
all: main.o hello.o
        gcc main.o hello.o -o hello

main.o: main.c hello.h
        gcc -c main.c

hello.o: hello.c hello.h
        gcc -c hello.c

clean:
        rm -f *.o hello
```

With this Makefile, to re-compile the whole program, we call “make all”. To remove all of object files and binaries, we call “make clean”. If we need to create an object file - main.o, we call “make main.o”. If we only call “make”, the default rule of Makefile is executed - “make all”.

References

- Coding style by GNU: `http://www.gnu.org/prep/standards/standards.html`.
- C programming
  - Brian Kernighan, and Dennis Ritchie, *"The C Programming Language"*, Second Edition
  - Randal E. Bryant and David R. O'Hallaron, *"Computer systems: A Programmer's Perspective"*, Second Edition
- More information about Vim: `http://vim.wikia.com/wiki/Vim_Tips_Wiki`
- Makefile:
  - A simple Makefile tutorial `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`
  - GNU Make Manual `https://www.gnu.org/software/make/manual/make.html`

3. Exercises

3.1. Questions

1. Briefly explain each stage of the compilation process.
2. Illustrate the compilation process when we have multiple source files. Ex:
   - main.c
   - transmit.c
   - transmit.h
   - receive.c
   - receive.h
3. Compiling a program in the first time usually takes a longer time in comparison with the next re-compiling. What is the reason?
4. How do we use Makefile for other programming languages? Use specific examples to support your answer.
5. When source code files are located in different directories, how can we write a Makefile?

3.2. Programming exercises

Two header files named findsubstr.h and readline.h have the following contents:

```c
// findsubstr.h
#ifndef FIND_SUBSTR_H
#define FIND_SUBSTR_H

int find_sub_string(const char *str, const char *sub);

#endif
```

```c
// readline.h
#ifndef READ_LINE_H
#define READ_LINE_H

int read_line(char *str);

#endif
```

1. Write findsubstr.c implementing find_sub_string(char *str, char *sub) function: **find_sub_string** gets two strings including **str** and **sub**. If **str** contains the substring (substr) that is similar to **sub**, the program returns the first letter of substr in **str**. Otherwise, the program returns -1. For example, find_sub_string("abcbc", "bc") returns 1, while find_sub_string("abcbc", "xy") returns -1. What is the complexity of your method? NOTE: bonus points if you can devise a O(N) algorithm.

2. Write readline.c implementing read_line(char* str): read_line() function gets data from stdin (keyboard), line-by-line. The content from stdin will be recorded to the parameter of this function named **str**. The result of read_line() is the number of read letters, or -1 if it reads EOF (end of file). For example, with the input string below:

```
Hello, world
Operating system
Computer Science and Engineering
```

After calling the function, read_line() writes "Hello,world" into str and returns 12. The second call will write "Operating system" into str and return 16. The third call will write "Computer Science and Engineering" into str and return -1.

3. Write main.c to develop an executable file named *mygrep* that performs similarly to the *grep* command on Linux/Mac OS X shell. Specifically, mygrep gets input being a string, and then reads one line of stdin at a time (each line does not exceed 100 letters). After that, the program checks which line contains the string being entered by user, and prints these lines on screen. After writing main.c file, write a Makefile to compile the program at least two targets:

- all: create mygrep from other files.
- clean: remove all of object files, binaries.

```
// main.c
#include <stdio.h>
#include "readline.h"
#include "findsubstr.h"

int main(int argc, char * argv[]) {
        // Implement mygrep
}

#endif
```

The syntax of **mygrep** will be:
./mygrep <input_string>

TESTING: To easily evaluate your program, create a text file input.txt like this:

```
Hello, world
Operating system
Computer Science and Engineering
```

Then execute your program like this:

```
cat input.txt | ./mygrep system
```

The output should be:

```
Operating system
```

NOTE: As these exercises are graded automatically, thereby, student need to implement the program by the requirements mentioned above. Compress all of source files (.c, .h, Makefile) together with a report in a folder named Student ID by .zip format. For example: 1510000.zip

A. General purpose Makefile

```
FC=gfortran
CC=gcc
CP=cp

.PHONY: all clean

OBJS = mylib.o mylib_c.o

# Compiler flags
FFLAGS = -g -traceback -heap-arrays 10 \
         -I. -L/usr/lib64 -lGL -lGLU -lX11 -lXext

CFLAGS = -g -traceback -heap-arrays 10 \
         -I. -lGL -lGLU -lX11 -lXext

MAKEFLAGS = -W -w

PRJ_BINS=hello
PRJ_OBJS = $(addsuffix .o,$(PRJ_BINS))

objects := $(PRJ_OBJS) $(OBJS)

all: myapp

%.o: %.f90
        $(FC) -D_MACHTYPE_LINUX $< -c -o $@

%.o: %.F
        $(FC) -D_MACHTYPE_LINUX $< -c -o $@

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

myapp: objects
        $(CC) $(CFLAGS) $^ $(objects) -o $@

clean:
        @echo "Cleaning␣up.."
        rm -f *.o
        rm -f $(PRJ_BINS)
```

B. Vim

Vim is the editor of choice for many developers and power users. It‘s a “modal” text editor based on the vi editor written by Bill Joy in the 1970s for a version of UNIX. It inherits the key bindings of vi, but also adds a great deal of functionality and extensibility that are missing from the original vi.
Vim has two modes for users:

- Command mode: allows user to do functions such as find, undo, etc.
- Insert mode: allows user to edit the content of text.

To turn the *Insert* mode to *Command* mode, we type ESC key or $Crtl-C$. Otherwise, to enter the *Insert* mode, type i or I, a, A, o, O. Some of basic commands in *Vim*:

- Save: enter :w
- Quit without Save and discard the change: enter :q!
- Save and Quit: enter :wq
- Move the cursor to the top of file: gg
- Move to the bottom: G
- Find a letter/string by going forward: enter /[letter/string] <Enter>
- Find a letter/string by going backward: enter ?[letter/string] <Enter>
- Repeat the previous finding: enter n
- Repeat the previous finding by going backward: enter N
- Delete a line: enter dd
- Undo: enter u
- Redo: enter Ctrl-R

Furthermore, *Vim* has a mode called “visual” that allows user to chose a paragraph for copying or, moving. To enter this mode, we need to turn the editor into *Command* mode and press “v”. After that, user use “arrow” keys to chose the paragraph, and then use the following commands:

- Copy: enter y
- Cut: enter d
- Paste: enter p

Revision History

Revision	Date	Author(s)	Description
1.0	11.03.15	PD Nguyen	created
1.1	11.09.15	PD Nguyen	add introduction and exercise section
2.0	25.02.16	PD Nguyen	Restructure the content to form an tutorial
2.1	20.08.16	DH Nguyen	Update C and Vim to Appendix
2.2	15.03.18	Truc Nguyen	Update for Spring 2018