HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 2
C Programming on Linux, Mac OS X
Course: Operating Systems

Nguyen Minh Tri
Email: nmtribk@cse.hcmut.edu.vn

January 22, 2018

Goal: The lab helps student to

- Review basic shell commands and get familiar with *vim* on Linux, Mac OS X.
- Review C programming with compiling and running a program on Linux, Mac OS X.

Content:

- Get familiar with *vim* - text editor.
- Programming with C language.
- Compile a program with Makefile.

Result:

- Able to program without GUI on Linux/Mac OS C by *vim*.
- Able to compile and run a program using Makefile.

1 Introduction

1.1 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

1.2 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: It is important to understand that why some computer languages (e.g. Scheme or Basic) are normally used with an interactive interpreter (where you type in commands that are immediately executed). C source codes are always compiled into binary code by a program called a "compiler" and then executed. This is actually a multi-step process which we describe in some detail here.

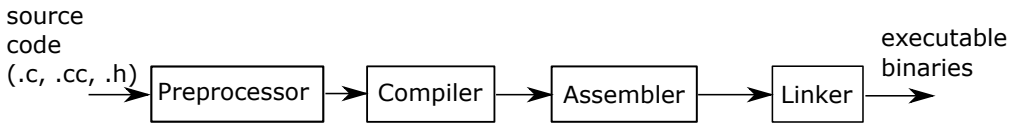

Figure 1.1: C Compiling scheme

Steps in compiling process:

- Preprocessing
- Compilation
- Assembly
- Linking

Preprocessor: A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

COMPILER: A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). Compilers are a type of translator that support digital devices, primarily computers. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program

BONUS: What is Interpreter? what is the different between compiler and interpreter?

ASSEMBLER: An assembler is used to translate the assembly instructions to machine code, or object code. The output consists of actual instructions to be run by the target processor.

LINKER: The linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces containing the instructions for library functions used by the program. The object code generated in the assembly stage is composed of machine instructions that the processor understands but some pieces of the program are out of order or missing. To produce an executable program, the existing pieces have to be rearranged and the missing ones filled in. This process is called linking.

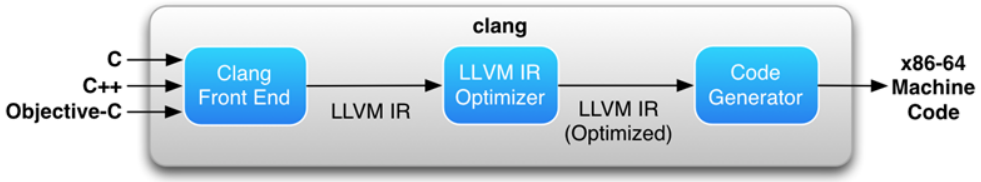

Figure 1.2: Clang in Mac OS X

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

```
1  % Preprocessed source file
2  $ gcc -E [-o hello.cpp] hello.c
3
4  % Assembly code
5  $ gcc -S [-o hello.S] hello.c
6
7  % Binary file
8  $ gcc -c [-o hello.o] hello.c
9
10 % Executable file
11 $ gcc [-o hello] hello.c
```

2 Practice

2.1 Compile and run a program

Steps for creating 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. Create object file:

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

3. Create 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 creating object file. However, this way can cause the difficulty when identifying errors.

4. Create 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 that show the information of errors. 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 Review 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. Compiling a program in the first time usually takes a longer time in comparison with the next re-compiling. What is the reason?
2. Is there any Makefile mechanism for other programming languages? If it has, give an example?
3. In case of source code files located in different places, how can we write a Makefile?

3.2 Programming exercises

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

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

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

#endif
```

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

int read_line(char *str);

#endif
```

1. Writing findsubstr.c implementing find_sub_string() 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.

2. Writing readline.c implementing read_line(): read_line() function gets data from stdin (keyboard), line-by-line. The content from stdin will be recorded on 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 calling will write “Operating system” into str and return 16. The third calling will write “Computer Science and Engineering” into str and return -1.

3. Writing main.c to create an executable file named mygrep that has function similar to grep command on Linux/Mac OS X shell. In detail, mygrep gets the input being a string, and then reads one line of strdin 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. Student finish main.c file, then 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
```

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

Makefile example

```
1  FC=gfortran
2  CC=gcc
3  CP=cp
4
5  .PHONY: all clean
6
7  OBJS = mylib.o mylib_c.o
8
9  # Compiler flags
10 FFLAGS = -g -traceback -heap-arrays 10 \
11          -I. -L/usr/lib64 -lGL -lGLU -lX11 -lXext
12
13 CFLAGS = -g -traceback -heap-arrays 10 \
14          -I. -lGL -lGLU -lX11 -lXext
15
16 MAKEFLAGS = -W -w
17
18 PRJ_BINS=hello
19 PRJ_OBJS = $(addsuffix .o,$(PRJ_BINS))
20
21 objects := $(PRJ_OBJS) $(OBJS)
22
23 all: myapp
24
25 %.o: %.f90
26          $(FC) -D_MACHTYPE_LINUX $< -c -o $@
27
28 %.o: %.F
29          $(FC) -D_MACHTYPE_LINUX $< -c -o $@
30
31 %.o: %.c
32          $(CC) -D_MACHTYPE_LINUX  $< -c -o $@
33
34 myapp: objects
35          $(CC) $(CFLAGS) $^ $(objects) -o $@
36
37 clean:
38          @echo "Cleaning up.."
39          rm -f *.o
40          rm -f $(PRJ_BINS)
```

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	20.01.17	MT Nguyen	Update C programming