

Lab 2

C Programming on Linux, Mac OS X

Course: Operating Systems

Toan V. Tran, Pham Trung Kien, Phuong-Duy Nguyen, Duc Hai Nguyen
Email: viettoantran98@gmail.com

March 29, 2021

Goal: The lab helps student to

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

Content:

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

Result:

- After doing the lab, student can type a program without GUI on Linux/Mac OS X by *vim*.
- Student can 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 *Ctrl-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 while 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.

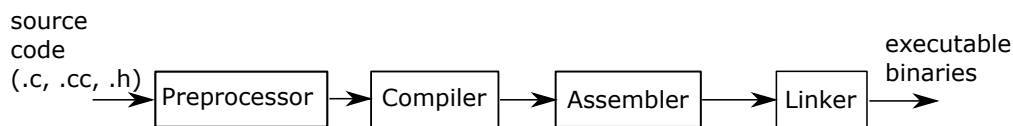


Figure 1.1: C Compiling scheme

STEPS IN COMPILING PROCESS:

- 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.

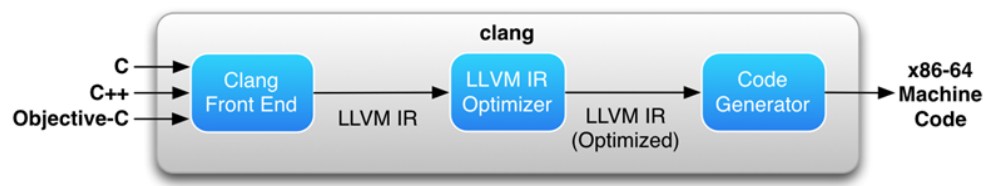


Figure 1.2: Clang in Mac OS X

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

```

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

```
#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 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 hello.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:

```
# File: 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. What are the advantages of Makefile? Give examples?
2. Is there any other automatic compilation tool besides Makefile? Give some names? What are the reasons for creating new compilation tools?
3. In case of source code files located in different places, how can we write a Makefile?

3.2 PROGRAMMING EXERCISES

Build your own shell. This exercise requires to write a C program that simulate a simple shell. To finish that, you are provided a source code and required to write a Makefile and fill your code inside "my_shell.c".

Ex 1) Write a Makefile that can compile the source. After running "\$ make all" the source must be compiled **to a binary file named "myShell"**. The source includes "parse_command.c", "parse_command.h", "my_shell.c". "my_shell.c" contains the main function. In addition, "parse_command.h" declares a function which is used in "my_shell.c".

To test your make file:

```
$ ./myShell
My Shell>$
My Shell>$ abc xyz
found token: abc
found token: xyz

My Shell>$
# You can use the shortcut of CTRL+D to exit.
$
```

Ex 2) After testing the Makefile, you must **comment two lines in "my_shell.c"**. Ex 2 requires you to fill the unfinished part in "my_shell.c".

Your task is to process tokens returned from parsing the command. For example, when:

```
$ ./myShell
My Shell>$ abc xyz
# The pointer named "tokens" will point to ["abc", "xyz"]
My Shell>$
```

In this task, you will implement commands:

1. whoami – this command will print your id

```
$ ./myShell
My Shell>$ whoami
```

```
My Id is 1911111
My Shell>$
```

2. counter – this command will print "Counter has been called n times!". In which, n start at 1 and will be increased by one if this command is called.

```
$ ./myShell
My Shell>$ counter
Counter has been called 1 times
My Shell>$ counter
Counter has been called 2 times
My Shell>$ counter
Counter has been called 3 times
```

3. quit – that will exit your shell without printing anything (similarly when CTRL+D)
4. simple version of system calls including "echo", "cat", "sleep", and "ls". You can follow some examples:

```
$ ./myShell
My Shell>$ echo my_text 123
my_text 123
My Shell>$ cat file.txt
content of file.txt
My Shell>$ sleep 3
My Shell>$ ls
parse_command.h parse_command.c my_shell.c myShell
```

Do not use the system() function to call system calls. We assume that all input commands are correct, so you do not to handle exceptions.

NOTE: As these exercises are graded automatically, thereby, you need to implement the program by the requirements mentioned above. you must compress all of files (.c, .h, Makefile) into a zip file as:

```
MSSV.zip
├── Question
│   └── answer.txt
└── Program
    ├── Makefile
    ├── my_shell.c
    ├── parse_command.h
    └── parse_command.c
```

Any cheating code detected will get 0 score. Students are required to show their running program during lab hours.

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
3.0	12.02.19	TK Pham	Change Exercises, edit Makefile explain
4.0	03.29.21	Toan VT	Change Exercises