ECE 264: Advanced C Programming

Lecture Notes 6/14/19

1 Topics

We did a review of the background topics necessary for programming assignment 1.

1. alias command
2. directives for conditional compilation
3. makefiles

2 The `alias` command

The alias command is used for creating shortcuts for commands that are long to type and are used often.

```
bash-4.1$ alias gcc='gcc -std=c99 -g -Wall -Werror -pedantic -Wvla -Wshadow'
```

The above command creates a shortcut for the `gcc` command typed with various flags shown. Next time, when `gcc` is used to compile a file `testgen.c` as:

```
bash-4.1$ gcc testgen.c -o testgen
```

this is equivalent to typing:

```
bash-4.1$ gcc -std=c99 -g -Wall -Werror -pedantic -Wvla -Wshadow testgen.c -o testgen
```

Note that a Makefile understands nothing about aliases. So, when you have a makefile with macro definitions and a rule as:

```
GCC=gcc
CFLAGS= -std=c99 -g -Wall -Werror --pedantic -Wvla -Wshadow
testgen:testgen.c
	$(GCC) $(CFLAGS) testgen.c -o testgen
```

`CFLAGS` stil needs to be defined in the Makefile.

The alias command is not preserved across terminal or login sessions i.e. when you open a new terminal and type `gcc`, the aliased version is not used. The unaliased version (just the `gcc` command without the flags) is used. In order to preserve aliasing across sessions, we need to include the alias command inside a `.bash_profile` file.

The `.bash_profile` file (if it exists) is usually located at your home directory (type the `cd` command and hit enter. Now type the command `pwd` to know the home directory). Edit the file to include the line:

```
alias gcc='gcc -std=c99 -g -Wall -Werror -pedantic -Wvla -Wshadow'
```

into the `.bash_profile` file. Next time, when you logout and login, all the commands listed in the `.bash_profile` are executed and the results are made available to you when you open a terminal. If the `.bash_profile` does not exist, create it in your home directory.

Another way to force the execution of all commands in the `.bash_profile` file without logging out and loggin back in is to execute the `source` command:

```
bash-4.1$ source .bash_profile
```

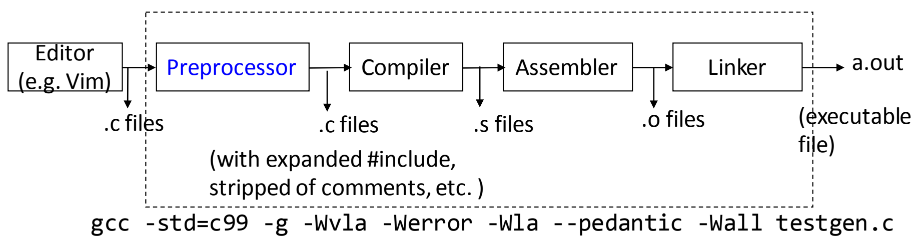
3 Conditional compilation

There are 6 directives and an operator `‘defined’` to write code for conditional compilation.

directives The directives are:

```
#if
#ifdef
#ifndef
#else
#elif
#endif
```

The directives are commands for the C *preprocessor*. Preprocessor is a piece of software, just like a compiler, that does some preprocessing on your source code (.c and .h files). Preprocessing is the first step in transforming your source code into an executable. The preprocessor transforms an input source code file into an output source code file after going through all preprocessor directives that exist in the file. A line of source code having a preprocessor directive begins with the `#` character. Common examples of preprocessor directives are `#define, #include, #pargma` etc. Below is a figure that shows different modules involved in transforming your source code into an executable.



So, when you type:

`bash-4.1$ gcc -std=c99 -g -Wall -Werror -pedantic -Wvla -Wshadow testgen.c -o testgen`

different modules work in the background to transform testgen.c into the executable testgen.

The conditional compilation directives tell the preprocessor to turn-on/off sections of code.

```c
//testpre.c
#ifdef DEBUG
void MyDebugFunction()
{
 //some block of code here used only while testing..
}
#endif

int main(){
#ifdef DEBUG
 printf("ECE264\n")
 MyDebugFunction();
#endif
}
```

if `MyTestFunction` is a large function containing few hundreds of lines of code and used only while debugging, we may not wish to have the function as part of the final binary released for production. We can use the C preprocessor directives to turn-off this section of the code. The Section 2.3 in the background topic in README.pdf for homework assignment 1 mentioned one way to enable/disable (turn-on/off) sections of your code enclosed in conditional compilation directives (using the `-D` flag along with the `gcc` command; as in `gcc -DDEBUG testpre.c -o testpre`). Another way to achieve the same effect is using the preprocessor directive `#define`. For example:

inserting the line:

`#define DEBUG`

at the beginning of the file `testpre.c` and then executing the command (note the absence of the -DDEBUG flag):

`bash-4.1$ gcc testpre.c -o testpre`

would compile the code in lines 3 to 6 and 11 to 12.

We could also achieve the same effect using the `#if` directive. Unlike `#ifdef`, where an identifier followed (e.g. `#ifdef DEBUG`), `#if` is followed by a constant-expression. Example: `#if 0, #if VERSION > 4`

```
1  //testpre.c
2  #define DEBUG 0
3  #if DEBUG
4  void MyDebugFunction()
5  {
6   //some block of code here used only while testing..
7  }
8  #endif
9
10 int main(){
11 #if DEBUG
12  printf("ECE264\n")
13  MyDebugFunction();
14 #endif
15 }
```

In the above version of `testpre.c` lines 4 to 7 and 12 to 13 would not be compiled. When we modify line 2 as `#define DEBUG 1` or `#define DEBUG 1234` we would see those lines getting compiled with an error (exercise: can you spot the error?).

Note that the directive `#ifdef` does not care about the value of the identifier (whether DEBUG is 0 or 1, or 1234). All it cares is that the identifier DEBUG is defined at the point where a piece of code is enclosed with the directive (line 2 in earlier version of `testpre.c`). The `'defined'` operator is used with `#if` as follows:

`#if (defined(VERSION1) && defined(VERSION3))`

The above directive tells that the following block of code is compiled only when both VERSION1 and VERSION3 are defined (again, the specific values of VERSION1 and VERSION3 don't matter.).

The other directives `#elif, #else` are used to compile sections of code using branching like a `if-else if-else` block in a `.c` file works.

4 Makefiles

We used the Makefile provided with homework assignment 1 (PA01) and went through a demo. As part of PA01 source files, we have provided `testgen.c` that creates several files with names `input`*xx* upon compiling and executing. The output files produced basically contain numbers to be sorted. Each of these files is used as an argument (on the command-line) to the other executable created `pa01`. `pa01` is the executable containing your sorting implementation in addition to other modules such as `swap, printarray, quicksort, etc.`.

Here is an overview of the executables/intermediate files (`.o` files) produced and the dependent files from which they are produced.

1. testgen.c -> testgen
2. pa01main.c -> pa01main.o
3. selectsort.c -> selectsort.o
4. quicksort.c -> quicksort.o
5. swap.c -> swap.o
6. printarray.c -> printarray.o
7. pa01main.o + selectsort.o + quicksort.o + swap.o + printarray.o -> pa01

In our demo, we saw the following:

1. how the first rule (`pa01`) gets fired (`bash-4.1$ make`)
2. what are the dependent rules to the first rule (`testgen` and `$(OBJS)`)
3. how firing the first rule takes us to the rule `testgen` followed by rules associated with `pa01main.o`, `selectsort.o` and other rules defined in `$(OBJS)`
4. how to fire the rule `testgen` explicitly (`bash-4.1$ make testgen`)
5. Not all rules defined in the Makefile are used to create pa01 (`testsome, testall, inspect, clean` are not fired in the process of executing the rule `pa01`).

exercise: figure out why the other rules not used in creating the executable `pa01` exist in the Makefile.

Note: `make clean` deletes all intermediate files created including `input`*xx*, `*.o` and others. It does not delete the changes done within your .c files.