Crash Course on UNIX and Systems Tools

Day 4 --- More Project Structure

Correction from Day 3

“Regular expression for positive integers”

- `^[0-9]+$` --- matches any digit **at least once** (b/c of ‘**+**’)
- `^[1-9][0-9]+$` --- a two-digit positive integer (**need one** to match `[1-9]` and **need at least another** to match `[0-9]+`)
- `^[1-9]+[0-9]*$` --- ‘*****’ indicates **zero** or more matches
- `^[1-9][0-9]*$` --- same as above, but the ‘**+**’ is not required
- Useful --- https://regex101.com/

Correction from Day 3

“Switching **`vim`** panels on Windows without **`ctrl-w`**” --- https://vi.stackexchange.com/questions/3728/how-can-i-work-with-splits-in-vim-without-ctrl-w

- **`ctrl-w`** can be executed with **`:winc`** (**`ctrl-w w`** == **`:winc w`**)
- You can also rename shortcuts (see link)

Project Structure: *Version control*

- **Initialize** a **repository** --- go to/build a directory and:

```
$ mkdir test ; cd test ; git init .
Initialized empty Git repository in
/root/sandbox/test/.git/
```

- We'll come back to this after modifying some code ...

Project Structure: *Version control*

We have some code --- **let's track it**

- Understand what files have changed in your repo:

`git status` (alternatively **`git diff`** to view those changes)

- Determine the structure for your **commit** --- what do you want to "take a **snapshot**" of or **track**? Stage those changes:

`git add lib.c`

Project Structure: *Version control*

- Take a look at what's ready to commit (again)

```
git status
```

- **Commit** your changes!
  - Choose a descriptive commit message
  - "Unstage" your changes if necessary --- go back to editing --- follow the directions from `git status`

```
git commit -m "library: add initial code"
```

Project Structure: *Version control*

- To see your commit history:

```
git log
```

- We’re on the master **branch** --- good practice to develop in a **separate branch** and leave the main line alone
  - `git branch -a` to view all your branches

```
git checkout -b dev
```

Project Structure: *Structuring your code*

- Create declarations for our simple math library (into **`lib.h`**)
- Separate the math library from the test (in **`main`** , into **`test.c`**)
- Within the code --- add **documentation**!
  - For others to understand the code
  - For you after you look at your code in 3 months
  - This code is documented for the purpose of the workshop

We've made more changes --- let's **commit** to our repository

Project Structure: *Makefiles*

We'll utilize makefiles from now on to compile our code. Why?

- Designed to facilitate application building and compilation
- Very easy to use --- **`make my_rule`**
- All you need to do is write the corresponding "**`./Makefile`**" and populate it with **rules**

Project Structure: *Makefiles*

- Makefile **rules** are “recipes” to build certain files or **targets** based on a set of commands and **dependencies**

```
my_executable: dependence.c
    gcc $^ -o $@
```

Project Structure: *Makefiles*

```
my_executable: dependence.c
    gcc $^ -o $@
```

- **`make my_executable`** will look for **`dependence.c`** first:
  - If **`dependence.c` exists**, the procedure continues
  - If **`dependence.c` does not exist** --- make looks for another rule to build dependence.c, but if there is none, the procedure fails
- **All dependencies** --- referred via the automatic variable **`$^`**
- **First dependence** --- referred via the automatic variable **`$<`**

Project Structure: *Makefiles*

```
my_executable: dependence.c
    gcc $^ -o $@
```

- **`make my_executable`** will attempt to build the **target `my_executable`**
  - If it already exists, **`make`** will **not rebuild**
  - The target can be referred via the automatic variable **`$@`**

Project Structure: *Makefiles*

- Variables in makefiles --- two main types (among others):
  - “**=**” sets a variable to a value ; whenever that variable is used, it’s expanded to its value **on demand**
  - “**:=**” also sets a variable to a value ; however, it expands the value **on assignment**
  - See https://stackoverflow.com/a/448973 for an example
- One way to refer to variables is as follows: **`$(MY_VAR)`**

Project Structure: *Makefiles*

Good idea to modularize your makefiles --- how?

- Variables for common values/commands --- some examples:
  - `CC:=gcc` for the C compiler
  - `CFLAGS:=...` for common flags to CC
  - etc.

Project Structure: *Makefiles*

- **`.PHONY`** targets --- Targets that aren't files/outputs to generate
  - **`make clean`** --- very common to remove the outputs/intermediate files, but doesn't generate a file called "clean"
  - **`.PHONY: clean`** --- to declare **`clean`** as a phony target
  - Consequences --- Acts as a **utility** command and you **can invoke it whenever you want** since the file "clean" never exists

Project Structure: *Separate compilation*

- Back to our code --- we have **several files** now, but **how do we compile** them? **Why** compile them using this method?
- **Separate compilation** --- two stage process:
  - Compile all **sources to object files** first, **link** them together **at the end**
  - Only sources with changes need to be recompiled an linked (makefiles facilitate this process)
  - Easier to test, easier to deploy parts of a large project, etc.

Project Structure: *Separate compilation*

- We’ll integrate this into our makefile
  - Subsequent slides will show the commands
  - The example makefile is at the end of the slides

Project Structure: *Separate compilation*

```
gcc -c lib.c -o lib.o
gcc -c test.c -o test.o
```

- `-c` : Compile but do not link

```
gcc lib.o test.o -o myprog
```

- Invoke the linking stage

Project Structure: *Separate compilation*

```
$ gcc lib.o test.o -o myprog
lib.o: In function `sqrt_of_factorial':
lib.c:(.text+0x7c): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

What do we do now?

Project Structure: *Separate compilation*

- Need to explicitly link the math library (**`libm`**) using the **`-lm`** flag
  - Note that the linker didn't complain about functions like **`printf`** because **`libc`** (the standard C library) automatically linked by **`gcc`**

```
gcc lib.o test.o -lm -o myprog
```

Project Structure: *Separate compilation*

```
gcc lib.o test.o -lm -o myprog
test.o:(.data+0x0): multiple definition of `N'
lib.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
Makefile:6: recipe for target 'myprog' failed
make: *** [myprog] Error 1
```

What now?

Project Structure: *Separate compilation*

- **`'N'`** has multiple definitions --- **`lib.h`** defines it (**`int N = 10`**), but it's included in each source file! That won't work
- Using **`extern`** for a global variable
  - A keyword that tells the compiler to look later and elsewhere for the value --- several declarations but one definition
  - In **`lib.h`** : **`extern int N;`**
  - In **`lib.c`** : **`int N = 10;`**
- Now we can compile successfully!

Project Structure: *Example makefile*

Here's where we are … we'll improve this on Day 5

```
CC:=gcc
CFLAGS:=-c
LIBS=:-lm

myprog: lib.o test.o
    $(CC) $^ $(LIBS) -o $@

lib.o: lib.c
    $(CC) $(CFLAGS) $< -o $@

test.o: test.c
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm -f lib.o test.o myprog

.PHONY: clean
```