Recitation 02

Preface

Preparing for Recitation 02

Pulling the second assignment

- Inside your recitations repository in your VM
  - Run **git remote add upstream** https://github.com/nyu-cso-fa19/recitations.git
    - You only do this once
  - Run **git pull upstream master**
    - You will do this for each new assignment

Pulling the second assignment

- If you see

```
lab@vbox:~/symmetrical-spoon$ git pull upstream master
warning: no common commits
remote: Enumerating objects: 68, done.
remote: Counting objects: 100% (68/68), done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 68 (delta 39), reused 64 (delta 37), pack-reused 0
Unpacking objects: 100% (68/68), done.
From https://github.com/nyu-cso-fa19/recitations
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> upstream/master
fatal: refusing to merge unrelated histories
```

Don’t panic!

- Run **git reset --hard upstream/master**
- Run **git push -f**

Today's agenda

- We will cover in recitation
  - Compiling with GCC
  - Makefiles
  - Testing code
    - Lab1 framework
- What you will do tonight
  - R02
    - Write a Makefile for part1
    - Write tests for part2

Compiling

The basics of GCC

What is a compiler?

- C code is for people, not computers
  - In fact, high level languages in general are for people
  - Computer processors only “understand” binary instructions
- A compiler translates code between languages
  - In our case, it translates from C (the source language) to machine code (the target language)
- An alternative way to do things is to have a program read the code and execute commands
  - Such a program is called an interpreter
  - Python is an example of a language that uses an interpreter

How do you use a compiler?

- Consider a simple C program:

main.c:

```
#include <stdio.h>
int main() {
        printf("Hello CSO!");
        return 0;
}
```

- To run this program, we must first compile it
  - We can use gcc: gcc main.c will produce a file called a.out
  - We can run a.out by issuing ./a.out
  - You can choose the name of the executable with -o, as in gcc main.c -o myprogram

How do you use a compiler?

- You can also compile more than one file

main.c:

```
void helper();
int main() {
        helper();
        return 0;
}
```

util.c:

```
#include <stdio.h>
void helper() {
        printf("Hello CSO!");
}
```

- To compile this, we can simply specify both files
  - gcc main.c util.c –o myprogram

A Problem

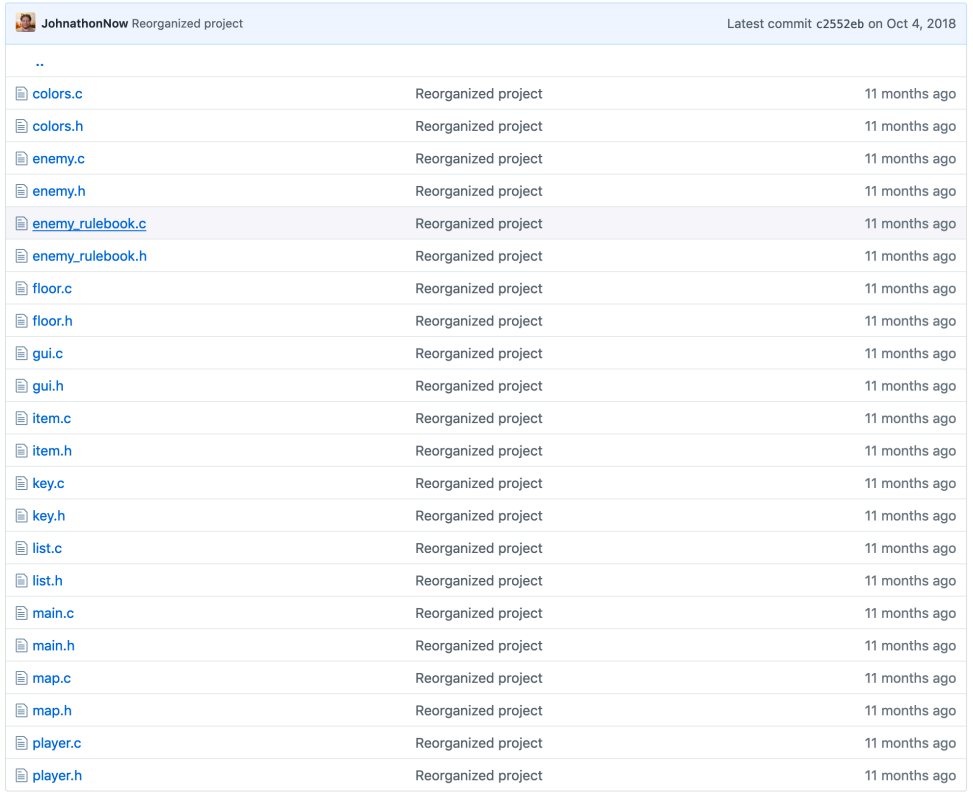
- For large projects, recompiling everything can be slow
- To avoid this, you can compile files separately
  - Note that you must compile without linking using the -c flag
  - Ordinarily, the compiler compiles each source file into object code, and then links them and deletes the intermediate object code
- gcc -c main.c util.c
  - Will create main.o and util.o
  - Then we can add to main.c or util.c and not have to recompile the other
  - We can later link by running gcc main.o util.o -o myprogram
- But now we need to keep track of when we have to recompile! >.<

Make

A helpful build automation tool

Why do we need Make?

- Even a small project like the one on the left is unbearable to compile with gcc alone
- But in the real world, things are much worse!
  - The Linux kernel has over **45,000** FILES of C code!
    - So it uses Makefiles… almost 2700 of them
- Make will also know when we need to recompile different sources



JohnathonNow Reorganized project		Latest commit c2552eb on Oct 4, 2018
..		
colors.c	Reorganized project	11 months ago
colors.h	Reorganized project	11 months ago
enemy.c	Reorganized project	11 months ago
enemy.h	Reorganized project	11 months ago
enemy_rulebook.c	Reorganized project	11 months ago
enemy_rulebook.h	Reorganized project	11 months ago
floor.c	Reorganized project	11 months ago
floor.h	Reorganized project	11 months ago
gui.c	Reorganized project	11 months ago
gui.h	Reorganized project	11 months ago
item.c	Reorganized project	11 months ago
item.h	Reorganized project	11 months ago
key.c	Reorganized project	11 months ago
key.h	Reorganized project	11 months ago
list.c	Reorganized project	11 months ago
list.h	Reorganized project	11 months ago
main.c	Reorganized project	11 months ago
main.h	Reorganized project	11 months ago
map.c	Reorganized project	11 months ago
map.h	Reorganized project	11 months ago
player.c	Reorganized project	11 months ago
player.h	Reorganized project	11 months ago

What does Make do?

- Make builds projects for us, keeping track of when it needs to recompile or not
- We tell make about the dependencies in our code using a Makefile
- Then, by issuing the command make we can build our project, and Make will only compile what it has to

What is a Makefile?

- Makefiles consist of a number of rules
- Rules specify:
  - A target, which is the thing we are trying to build
    - Targets include main.o or myprogram
  - Dependencies, which are what a target needs
    - main.o needs main.c
    - myprogram needs main.o and util.o
  - Commands, which are what create the target
    - gcc –c main.c –o main.o
- Running make builds the first target by default

What is a Makefile?

- Rules look like this:

myprogram: main.o util.o

    gcc main.o util.o –o myprogram

- There must be no space before the target, and there must be a tab before every command for that rule
- A bad Makefile for this little project is

myprogram: main.c util.c

    gcc main.c util.c –o myprogram

- Why is that bad?

A better makefile

```
myprogram: main.o util.o
	gcc main.o util.o –o my program

main.o: main.c
	gcc -c main.c –o main.o

util.o: util.c
	gcc -c util.c –o util.o

clean:
	rm –f main.o util.o myprogram
```

That still seems bad for the 45,000 linux files TA John

- That's right, and there are better ways of using Makefiles – this is just what you absolutely positively need to know
- Make also supports pattern matching with the percent sign %
  - %.c means all .c files
- Make has "automatic variables"
  - Variables whose meaning within a rule depends on context
  - $@ is the name of the rule
  - $^ is the list of dependencies

Testing

Making sure your code does what you think it does

Why test code?

- You need to know that your code works
- You need to know when you broke your own code by changing something
- Many projects actually have more test code than production code
  - An extreme example is SQLite, a popular database program
    - 138,900 lines of C code for production
    - 91,946,200 lines of test code

How do you test code?

- A common way is to write tests for individual units of code, such as functions
- There are many frameworks written to help developers write test cases
  - Wikipedia lists more than 50 for the C programming language
- You don't need a framework, though
  - You can write your own tests
    - Think of edge cases that might make your code sad
    - Write a program that calls your code with different inputs and checks that the output is what you'd expect
      - You can use assert to have your program die if something goes wrong
      - `assert(2+2==4)` will crash if 2 + 2 is not 4, but be fine otherwise