

# Computer System Organization Recitation [Spring 2019] CSCI-UA 0201-002

## R2: Compiling and Debugging

**Some slides based on Chien-Chin Huang's Spring 2018 CSO recitation**

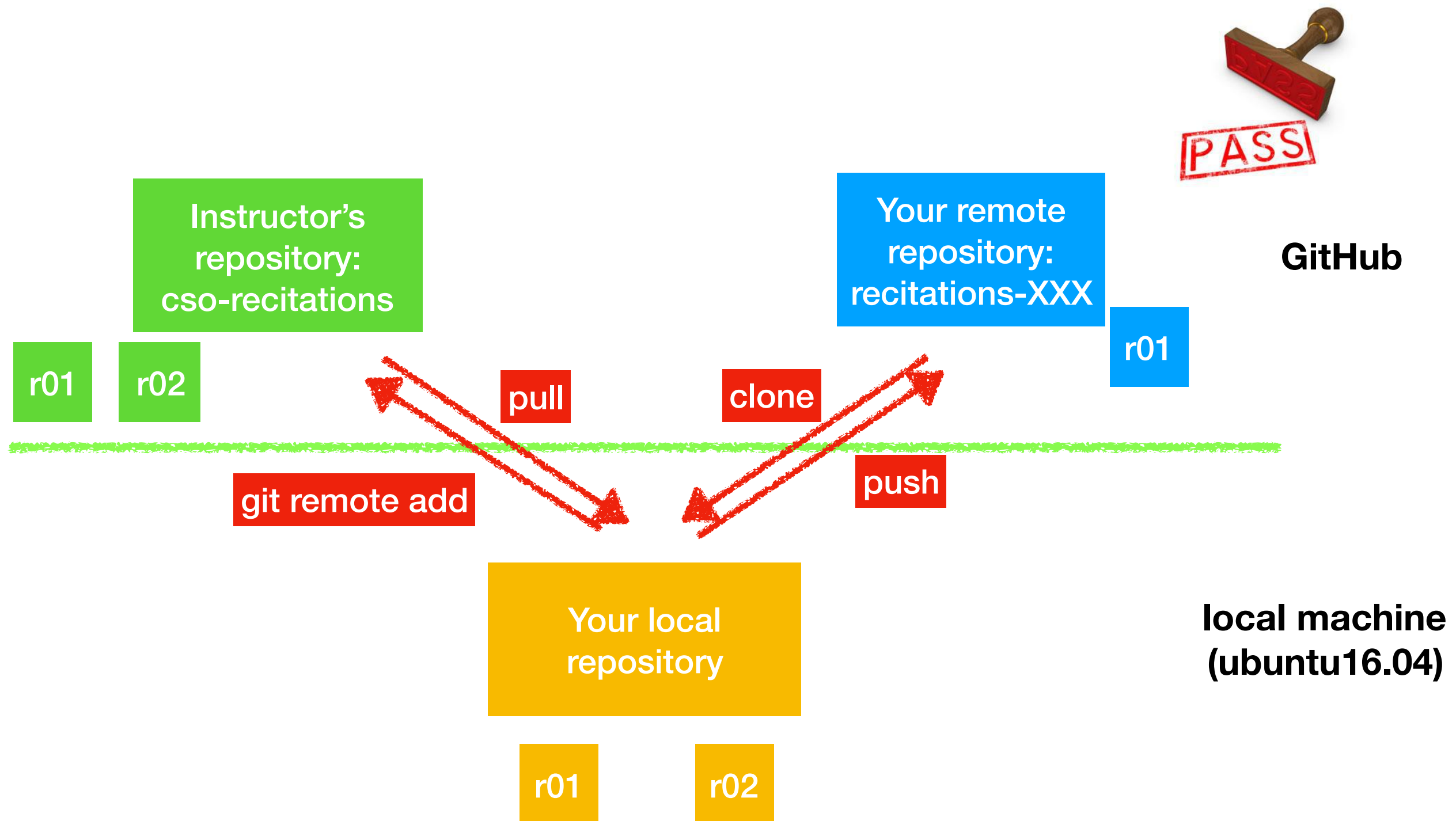
# Recitation README.md

- Under each recitation folder, there is a README.md, which contains contents, exercise instructions and deadline for each recitation
  - r01/README.md
  - r02/README.md
  - ...
- Always check the readme for each recitation
- Use GitHub to read README.md (and cheat sheet)
  - .md file is written in Markdown syntax
  - GitHub.com will render it properly

# How to sign CSO\_CHEAT\_SHEET

- Open the file for modification
  - Open up a terminal and go to r01 folder
  - `subl CSO_CHEAT_SHEET.md`
  - **Do NOT** make any changes on GitHub
- submit your changes
  - `git add CSO_CHEAT_SHEET.md`
  - `git commit -m "sign cheat sheet"`
  - `git push origin master`

# Work flow for our recitation and labs



# Track your instructor's repository

- For recitation repository:
  - `cd cso-recitations`
  - `git remote add upstream https://github.com/nyu-cso-sp19/cso-recitations`
- For lab repository:
  - `cd cso-labs`
  - `git remote add upstream https://github.com/nyu-cso-sp19/cso-labs`
- You only need to do this once

# For each lab and recitation

- pull latest lab and recitation materials
  - `git pull upstream master`
- then make changes locally on you computer
- tell git to track changes
  - `git add “file name”`
- commit changes
  - `git commit -m “commit messages”`
- submit to your remote repository
  - `git push origin master`

# Double check with “git status”

- git status tells you
  - what files are changed
  - what files are going to commit
  - what files are not tracked
  - whether your local repository and the remote repository is up-to-date

# Double check with “git status”

- If you changed the cheat sheet and readme file, and you only “git add” the cheat sheet

```
→ r01 git:(master) ✕ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   CSO_CHEAT_SHEET.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        r01-backup.key
```



# Triple check with GitHub

- Still not confident about whether assignment was submitted properly?
- Go to [github.com](https://github.com), and navigate to your repo
- Manually check if every file contains your latest changes

# Git is much more powerful than that

- Our git introduction only covers a small part of Git
- Git tutorial:
  - <https://www.atlassian.com/git/tutorials/what-is-version-control>
  - <https://try.github.io/levels/1/challenges/1>
- Please only test your skills elsewhere
  - Don't try advanced skills in your lab or recitation repo

# Common mistakes about Git

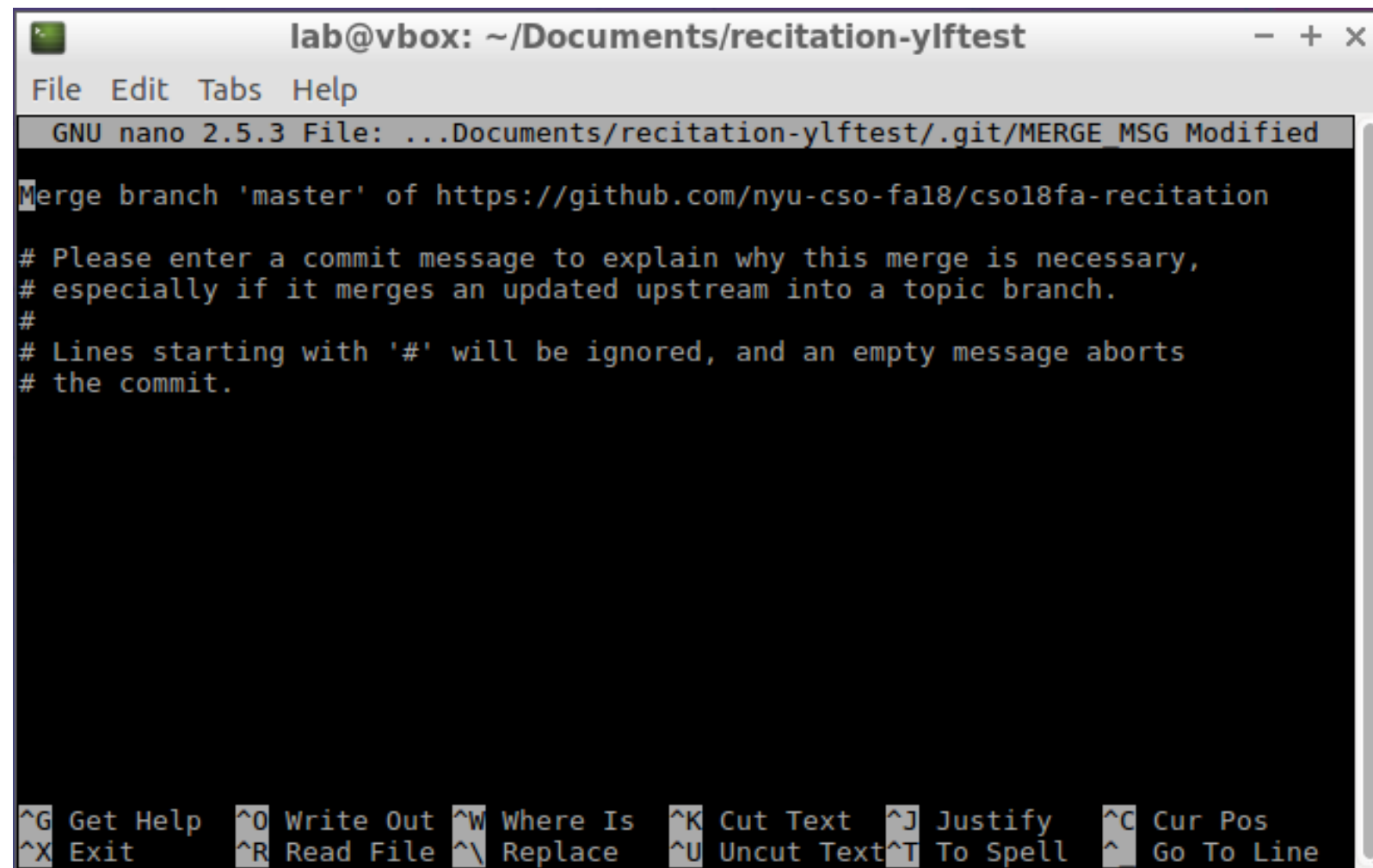
- Recitation and lab are two separate/independent repositories
  - Don't clone lab repository inside recitation repository, vice versa.
  - Don't try to merge them into one...
- You need to set up upstream correctly
  - check out instructions on CSO\_CHEAT\_SHEET.md
- Need help?
  - Come to office hour

# For recitation 01

- Change every XXX with your Github Username (not your name)
- Recitation 01 due Feb 6th 11 pm
- Double check with git status and triple check with Github

# Retrieve r02 from upstream

- git pull upstream master
- Get a merge message? Hit **ctrl + x**



The screenshot shows a terminal window titled "lab@vbox: ~/Documents/recitation-ylftest". The window contains the GNU nano 2.5.3 text editor. The file being edited is "...Documents/recitation-ylftest/.git/MERGE\_MSG Modified". The editor displays the following text:

```
Merge branch 'master' of https://github.com/nyu-cso-fa18/cso18fa-recitation

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

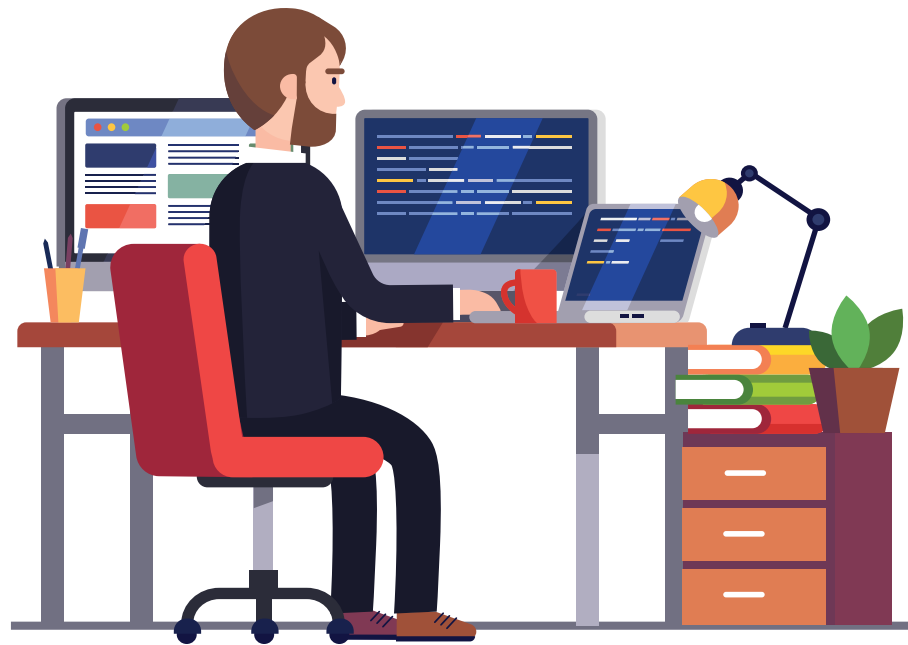
The bottom of the screen shows the nano editor's command shortcuts:

```
^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify    ^C Cur Pos
^X Exit      ^R Read File  ^\ Replace   ^U Uncut Text ^T To Spell   ^_ Go To Line
```

# Today's topic

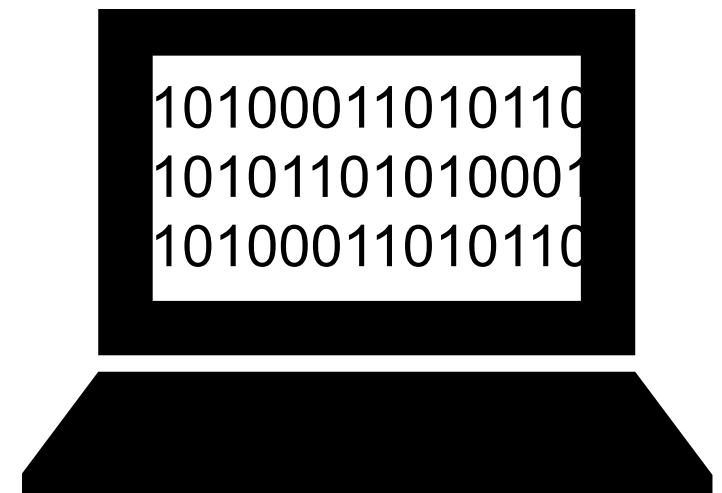
- Compilation with GCC
- Manage project compilation with make
- Debug code with GDB

# Compile



**You program in high  
level language**

**Compile**



**Machine only understands  
binary instructions**

**Images from <https://www.freepik.com/>**

# Terminology

- Compiler
  - A software that transforms computer code written in one programming language (the source language) into another language (the target language)
- Interpreter
  - A software that directly executes instructions written in a programming/scripting language without compiling them into a machine language program in advance



# Terminology

- source file
  - the input of compiler, where you write the program in high level language like C
- target file
  - the output of compiler
- executable (file)
  - file that can be executed by machine directly

# C, Python, and Java

- C language uses compiler to translate source file directly into machine code which is executable by your machine
- Python uses a interpreter to translate source code and execute line by line at runtime
- Java first compiles its source code to a more efficient immediate representation (called byte code) and uses a interpreter to run byte code

# Compiling v.s. interpreting

- Compiling
  - No translation needed at runtime (fast)
  - More optimization opportunities (efficient)
  - Compiler is usually able to detect many problems during compilation
- Interpreting
  - Flexible and more interactive: you can decide which code to execute at runtime
  - Requires a runtime software, which can also be used to:
    - hide platform details (code is more portable)
    - keep runtime information and perform runtime checks (like boundary check, garbage collection)
  - The price you pay is speed

# In CSO...

- We are going to use a language called C
  - It will be compiled directly into machine code
- We will use GCC as our C compiler

# GCC: basic usage

- You wrote your first hello\_world.c
- Compile it in terminal:
  - gcc hello\_world.c
  - an executable file “a.out” will be generated
- Run your program
  - ./a.out
- Compile your source code into a different name
  - gcc hello\_world.c -o hello\_world

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

hello\_world.c

# Compile multiple source files into one executable

- Code are placed in multiple files
- To compile multiple files into one executable

```
gcc foo.c main.c -o main
```

```
int foo() {  
    int i = 0;  
    i += 1;  
    return i;  
}
```

**foo.c**

```
int main() {  
    foo();  
    return 0;  
}
```

**main.c**

# Compilation

```
int foo() {  
    int i = 0;  
    i += 1;  
    return i;  
}
```

foo.c

compile



```
foo:  
...  
movl 0, %eax  
movl 1, %ebx  
addl %ebx, %eax  
...
```

foo.o

```
int main() {  
    foo();  
}
```

main.c

compile



```
main:  
...  
move foo, %eax  
call %eax  
...
```

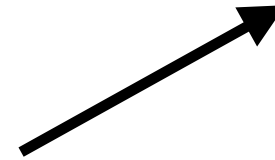
main.o

Keep the \*.o for  
fast recompiling

Linker

main

Temporary files, automatically  
removed after compiling



# Compilation

```
int foo() {  
    int i = 0;  
    i += 1;  
    return i;  
}
```

foo.c

compile



```
foo:  
...  
movl 0, %eax  
movl 1, %ebx  
addl %ebx, %eax  
...
```

foo.o

```
int main() {  
    foo();  
    Return 0;  
}
```

main.c

recompile



```
main:  
...  
move foo, %eax  
call %eax  
...
```

main.o



main



# To save intermediate files

- Compile without linking (-c option)

```
gcc -c foo.c
```

```
gcc -c main.c
```

- Run linker in a separate step

```
gcc -o main foo.o main.o
```

# Compile large projects

- If we have a very large project, containing **1 MILLION** files
  - How can we manually run gcc to compile files one by one?
  - How can we know which file is changed and needs recompilation?
  - How can we give different compiling options when compile different files?
  - How can we compile files in parallel using multiple CPU cores and figure out dependency correctly?

# You want one button to launch the rocket?



- Image from <https://www.canstockphoto.com/businessman-pushing-the-start-button-51114320.html>

# Manage compiling with Make

- What is Make?
  - A build automation tool that automatically builds executable programs and libraries from source code.
- How?
  - Describe compilation rules in Makefile to specify dependencies, and Make will do everything for you.
- The button to launch rocket
  - just type ``make``

# How to write Makefile

- Makefile consists of a bunch of rules

target: dependencies

commands

- The rule means if any file in dependencies changed, generate target using commands

For example:

main: main.c foo.c

gcc -o main main.c foo.c

# Simplest but terrible way to write Makefile

## Makefile

**main:**

**gcc -o main main.c foo.c**

File format

- No white space before “all”
- one tab before commands

- Type `make` and the command will be executed
- It's terrible because
  - you still handle compiling manually
  - Recompile everything if you type make again

# A good Makefile

## Makefile

```
main: foo.o main.o
    gcc -o main main.o foo.o
```

```
foo.o: foo.c
    gcc -c foo.c
```

```
main.o: main.c
    gcc -c main.c
```

```
clean:
    rm -f main main.o foo.o
```

- Make knows file dependency and can automatically figure out which files need recompilation
  - By checking file timestamp
- Problem:
  - What if I have 1M source files?

# Pattern matching and automatic variables

- Use pattern matching to define dependency
  - `%.o: %.c`
  - `%` matches the stem of file name
  - any target ends with `.o` depends on the file start with the same filename stem and ends with `.c`
- Use automatic variables to refer to target and dependency in commands
  - `$@` (target name)
  - `^` (name of all prerequisites, i.e. dependencies)
  - `<` (name of the first prerequisite)



# Here is a better one

## Makefile

```
main: foo.o main.o
```

```
gcc -o $@ $^
```

```
%.o: %.c
```

```
gcc -o $@ -c $<
```

```
clean:
```

```
rm -f main main.o foo.o
```

- Succinct enough
- Can automatically compile source file into object file
- Not general enough
  - If we have more files to compile, we have to change the first and last rules manually

# Let's make it even better

## Makefile

<b>SRC:=\$(wildcard *.c)</b>	<b>←</b>	<b>Find all files ending with .c</b>
<b>OBJ:=\$(SRC:.c=.o)</b>	<b>←</b>	<b>Replace .c with .o</b>

```
main: $(OBJ)
    gcc -o $@ $^
```

```
%.o: %.c
    gcc -o $@ -c $<
```

```
clean:
    rm -f main $(OBJ)
```

# The target “clean”

- If you type “make clean”, executable and immediate files will be deleted
- If you change the Makefile:
  - make clean
  - make
- Otherwise, make thinks everything is up-to-date because Makefile itself is not included in the dependency list

# Run the executable

- Now that you can compile your code, run it by type  
`./main`
- And you will see the program hangs there...
- Hit Ctrl+C to interrupt and kill the program
- Actually, most of the time, you will find yourself mostly writing bugs when you first start to program in C

# Debug

- How to debug your code?
  - The best way to debug is to write assertions, print logs (program states), observe and then debug
  - Use a debugger to help you
    - ▶ `gdb`
- What does a debugger do?
  - It executes program step by step in an interactive fashion
  - It allows you to examine program state whenever you want
- To use `gdb`
  - First ask `gcc` to add debug information when compiling the source files.

# Add debugging flag to Makefile

## Makefile

```
SRC:=$(wildcard *.c)  
OBJ:=$(SRC:.c=.o)
```

```
main: $(OBJ)  
    gcc -o $@ $^
```

```
%.o: %.c  
    gcc -o $@ -c $< -g
```

```
clean:  
    rm -f main $(OBJ)
```

# How to use gdb

- a list of commands you will frequently use
  - b (set break point)
  - r (run the program)
  - n (execute one statement, treat one function call as a step)
  - s (execute one statement, will go into function)
  - p (examine and print value of an expression)
  - l (print lines from the relevant source file)
  - c (continue execution)
  - q (quit debugging)

# To learn more about GDB

- Learning C with gdb
  - <https://www.recurse.com/blog/5-learning-c-with-gdb>
- A Youtube video on gdb
  - <https://www.youtube.com/watch?v=xQ0ONbt-qPs>



# What to submit

- Modified Makefile
  - use pattern matching and automatic variable
  - with debugging enabled
- Fixed foo.c
- Due Feb 6th 11 pm
- Don't forget to submit recitation 01 as well

# A few more words...

- We have talked about VM, bash, git, gcc, make, gdb
- If you feel you still understand nothing...
  - Don't panic too early
- You don't need to be a master these tools to pass CSO
  - They help you do coursework more smoothly
  - You will be using them for your entire life, so learn them early!
- learn them from online resources (especially those we provide to you)

# If you want to get a good grade for CSO

- I hope Professor has convinced you that CSO is pretty hard if you don't work hard
- Expect to devote **at least 15 hours** to CSO every week
  - Or you might easily get a D...
- If you think you are lagging behind, speak to the Professor for advice.
- Come to office hour
- Never wait until last second to ask for help or to submit your assignments