Makefiles, include guards, and Xv6

Dr. Naser Al Madi

Learning objectives

- Makefiles
- Include guards
- fork + wait + exit
- Go over Xv6 and qemu
- Write hello world in Xv6
- Write PS system call together

Installing C environment

*reminder

Install instructions

Mac

- Install VS Code
- Install the following extensions to VS Code:
    - C/C++ by Microsoft
    - Code Runner
- In a terminal, check if you have a c compiler:
    - clang –version
    - If not installed, type: xcode-select --install

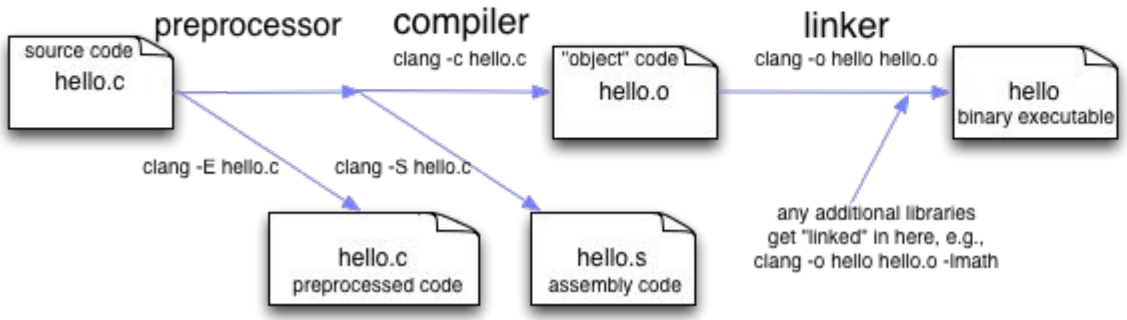
Check this page for troubleshooting.

Windows 10

- Install VS Code
- Install the following extensions to VS Code:
    - C/C++ by Microsoft
    - Code Runner
- Install Windows Subsystem for Linux (WSL)
    - Open PowerShell
    - wsl --install -d ubuntu
- Run Ubuntu
- Configure Ubuntu with username
- sudo apt-get update && sudo apt-get install git nasm build-essential qemu gdb

Check this page for troubleshooting

Compiling multiple files



Source: https://jsommers.github.io/cbook/programstructure

Multiple files

func1.h

```c
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

main.c

```c
#include “func1.h”

int main() {

  myFunction2();

  return 0;
}
```

Multiple files

func1.h

```c
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

main.c

```c
#include “func1.h”

int main() {

  myFunction2();

  return 0;
}
```

```
>gcc main.c
```

More interesting multiple files

func1.h

```
void myFunction2();
```

func1.c

```
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

main.c

```
#include “func1.h”

int main() {

  myFunction2();

  return 0;
}
```

```
>gcc main.c func1.c
```

Object files

func1.h

```
void myFunction2();
```

func1.c

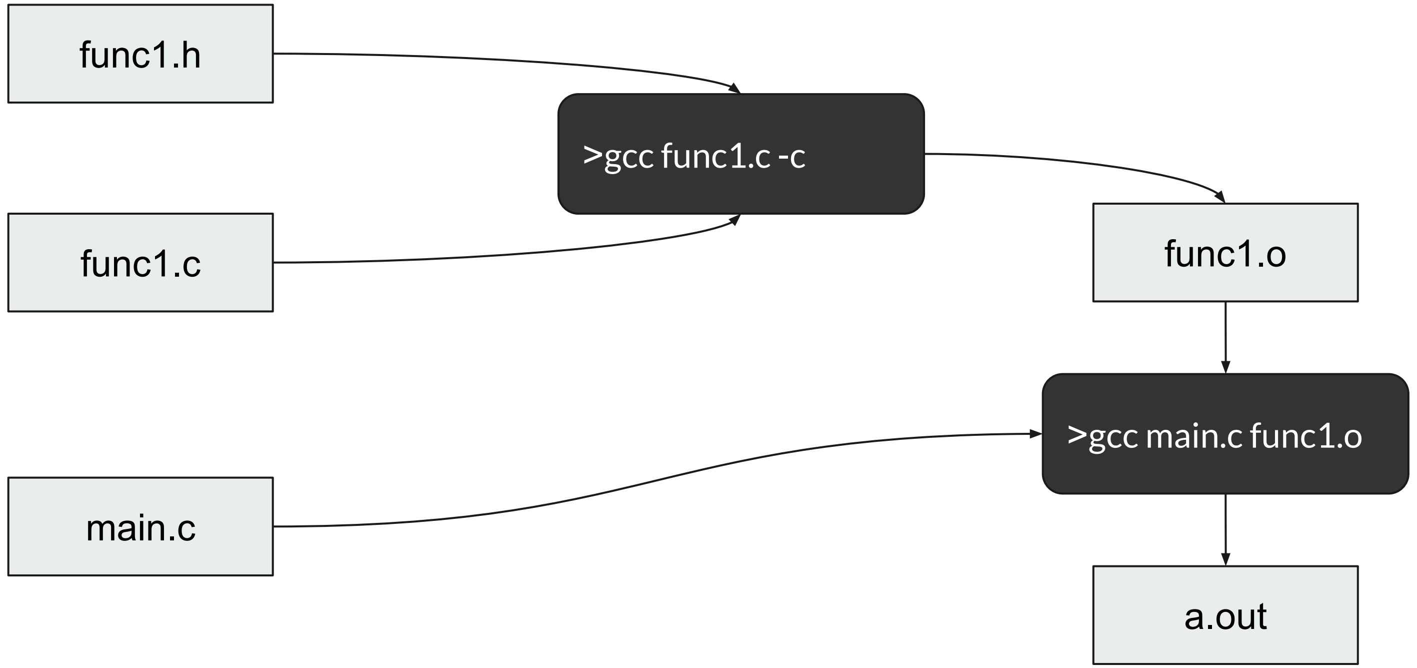
```
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

main.c

```
#include “func1.h”

int main() {

  myFunction2();

  return 0;
}
```

```
>gcc func1.c -c

>ls

>func1.h  func1.c  main.c  func1.o

>gcc main.c func1.o

>./a.out

>I just got executed!
```

Object files



```
>gcc func1.c -c
```

```
>gcc main.c func1.o
```

funcs.h

```c
void start();
void myFunction2();
```

func1.c

```c
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

func2.c

```c
#include <stdio.h>

void start() {
  printf("Starting!");
}
```

main.c

```c
#include “funcs.h”

int main() {
  start();
  myFunction2();

  return 0;
}
```

```
># how to compile this?
```

funcs.h

```c
void start();
void myFunction2();
```

func1.c

```c
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

func2.c

```c
#include <stdio.h>

void start() {
  printf("Starting!");
}
```

main.c

```c
#include “funcs.h”

int main() {
  start();
  myFunction2();

  return 0;
}
```

```
>gcc main.c func1.c func2.c
```

OR

funcs.h

```c
void start();
void myFunction2();
```

func1.c

```c
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

func2.c

```c
#include <stdio.h>

void start() {
  printf("Starting!");
}
```

main.c

```c
#include “funcs.h”

int main() {
  start();
  myFunction2();

  return 0;
}
```

```
>gcc -c func1.c

>gcc -c func2.c

>gcc main.c func1.o func2.o
```

funcs.h

```c
void start();
void myFunction2();
```

func1.c

```c
#include <stdio.h>

void myFunction2() {
  printf("I just got executed!");
}
```

func2.c

```c
#include <stdio.h>

void start() {
  printf("Starting!");
}
```

main.c

```c
#include “funcs.h”

int main() {
  start();
  myFunction2();

  return 0;
}
```

```
>gcc -c func1.c

>gcc -c func2.c

>gcc main.c func1.o func2.o
```

Makefiles

A makefile is a file (by default named "Makefile") containing a set of directives used by a make build automation tool to generate a target/goal. [From Wikipedia]

Makefiles

Each rule follows the logic:

```
target … : prerequisites …
      recipe
      …
      …
```

Makefiles

Things to remember:

- If no rule specified make will execute the first rule in the makefile.
- Rules are executed recursively until all dependencies are created.
- Tabs and not spaces before command (recipe)

Makefiles reference slide

There are seven “core” automatic variables:

- $@: The filename representing the target.
- $%: The filename element of an archive member specification.
- $<: The filename of the first prerequisite.
- $?: The names of all prerequisites that are newer than the target, separated by spaces.
- $^: The filenames of all the prerequisites, separated by spaces. This list has duplicate filenames removed since for most uses, such as compiling, copying, etc., duplicates are not wanted.
- $+: Similar to $^, this is the names of all the prerequisites separated by spaces, except that $+ includes duplicates. This variable was created for specific situations such as arguments to linkers where duplicate values have meaning.
- $*: The stem of the target filename. A stem is typically a filename without its suffix. Its use outside of pattern rules is discouraged.

Learn more: https://www.gnu.org/software/make/manual/html_node/index.html

Makefile

```
all:
	gcc test.c func1.c func2.c -o test
```

```
>make

>./test
```

Makefile

```
all:
	gcc test.c func1.c -o test
	./test
```

```
>make
```

Makefile

```
all:
	gcc test.c func1.c -o test
	./test

clean:
	rm -f test
	rm *.o
```

```
>make clean
```

Makefile

```
CC = gcc
CFLAGS = -Wall
OBJ = test


test: test.c func1.o
	$(CC)  $(CFLAGS) test.c func1.o -o $(OBJ)

func1.o: func1.c func1.h
	$(CC)  $(CFLAGS) -c func1.c

run: $(OBJ)
	./$(OBJ)

clean:
	rm -f test
	rm *.o
```

```
>make run
```

Bigint makefile in class activity!

Include guards

Include Guards

mystring.h

```c
int function(){

    return 5;
}
```

otherfile.h

```c
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```c
#include <stdio.h>
#include "mystring.h"
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

Include Guards

mystring.h

```c
int function(){

    return 5;
}
```

otherfile.h

```c
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```c
#include <stdio.h>
#include "mystring.h"
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

```
>clang main.c

./mystring.h:3:5: error: redefinition of 'function'
```

Include Guards

mystring.h

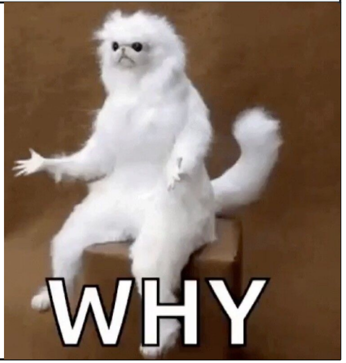
```c
int function(){

    return 5;
}
```

otherfile.h

```c
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

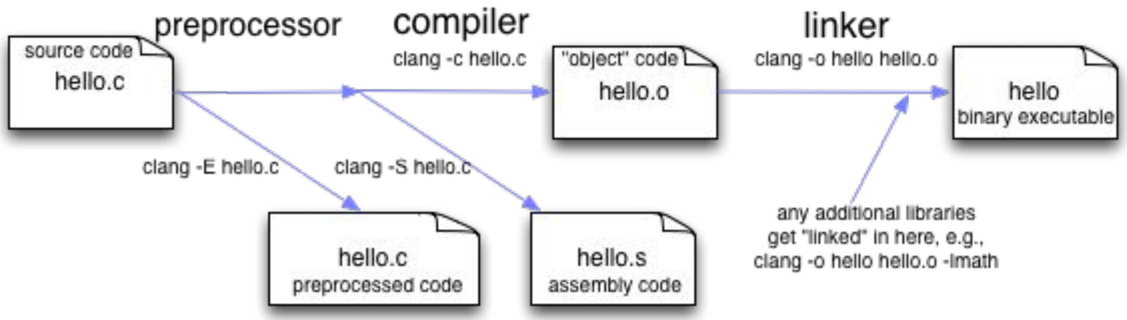
main.c

```c
#include <stdio.h>
#include "mystring.h"
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```



```
>clang main.c

./mystring.h:3:5: error: redefinition of 'function'
```

Here’s the problem



Source: https://jsommers.github.io/cbook/programstructure

Preprocessor

mystring.h

```
int function(){

    return 5;
}
```

otherfile.h

```
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```
#include <stdio.h>
#include "mystring.h"
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

Preprocessor

mystring.h

```
int function(){

    return 5;
}
```

otherfile.h

```
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```
#include <stdio.h>
#include "mystring.h"
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

Preprocessor

mystring.h

```c
int function(){

    return 5;
}
```

otherfile.h

```c
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```c
#include <stdio.h>
int function(){

    return 5;
}
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

Preprocessor

mystring.h

```
int function(){

    return 5;
}
```

otherfile.h

```
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```
#include <stdio.h>
int function(){

    return 5;
}
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

Preprocessor

mystring.h

```c
int function(){

    return 5;
}
```

otherfile.h

```c
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```c
#include <stdio.h>
int function(){

    return 5;
}
#include "mystring.h"

void somefunction(){
    int x = 10;
}

int main(){
    printf("Hello there!\n");
    return 0;
}
```

Preprocessor

mystring.h

```
int function(){

    return 5;
}
```

otherfile.h

```
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

main.c

```
#include <stdio.h>
int function(){

    return 5;
}
int function(){

    return 5;
}

void somefunction(){
    int x = 10;
}

int main(){
    printf("Hello there!\n");
    return 0;
}
```

Here's the Solution *Include Guard*

Include Guards

mystring.h

```
#ifndef MYSTRING_H
#define MYSTRING_H

int function(){

    return 5;
}
#endif
```

main.c

```
#include <stdio.h>
#include "mystring.h"
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

otherfile.h

```
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

Include Guards

mystring.h

```
#ifndef MYSTRING_H
#define MYSTRING_H

int function(){

    return 5;
}
#endif
```

main.c

```
#include <stdio.h>
#include "mystring.h"
#include "otherfile.h"

int main(){
    printf("Hello there!\n");
    return 0;
}
```

otherfile.h

```
#include "mystring.h"

void somefunction(){
    int x = 10;
}
```

We should add include guards to every header file!

Xv6

Let’s explore the make file and code of Xv6 together

https://github.com/nalmadi/xv6-public

Installing Xv6

Mac	Windows 10
• clone/download https://github.com/nalmadi/xv6-public • Install homebrew • brew install qemu x86_64-elf-gcc • export TOOLPREFIX=x86_64-elf- • export QEMU=qemu-system-x86_64 • Navigate to Xv6 directory • make • make qemu-nox	• Run Ubuntu • sudo apt-get update && sudo apt-get install git nasm build-essential qemu gdb • clone/download https://github.com/nalmadi/xv6-public • Open the makefile and add "-display none" to the end of the following line: QEMUOPTS = -hdb fs.img xv6.img -smp $(CPUS) -m 512 $(QEMUEXTRA) • Make • make qemu-nox