600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- Floats
- From code to executables
  - Header files
  - Independent compilation and linking
  - `make` and `Makefiles`

Integer value representation

Recall:

- On most machines, `int`s are represented using 4 bytes (32 bits)
  - These encode the integer in binary (mod 2^{32}), using 32 digits of precision

Floating point value representation

$$\pm b \times 2^{e}$$

- On most machines, `float`s are represented using 4 bytes (32 bits)
  - These are (roughly) used to encode:
    - The sign ($\pm$): 1 bit
    - The signed (integer) exponent (e): 8 bits*
    - The unsigned (integer) base (b): 23 bits

*This is what makes the point float

Floating point value representation

[WARNING]:

- Adding floating point values requires aligning their precisions first*

$$b_1 \times 2^{e_1} + b_2 \times 2^{e_2}$$
$$\Downarrow$$
$$(b_1 \times 2^{e_1 - e_2}) \times 2^{e_2} + b_2 \times 2^{e_2}$$
$$\Downarrow$$
$$(b_1 / 2^{e_2 - e_1} + b_2) \times 2^{e_2}$$

⇒ If b_1 is less than $2^{e_2 - e_1}$, then $b_1 / 2^{e_2 - e_1}$ will become zero

⇒ Addition of floating points may not associative:

$$(a + b) + c \; ! = a + (b + c)$$

*Assume $e_2 > e_1$

Floating point value representation

[WARNING]:

- Adding floating point values requires aligning their precisions first*

```
#include <stdio.h>
int main( void )
{
        float a = 1e-4f , b = 1e+4f , c = -b;
        printf( "%.3e %.3e %.3e\n" , a , b , c );
        printf( "%.3e %.3e\n" , ( a + b ) + c , a + ( b + c ) );
        return 0;
}
```

```
>> ./a.out
1.000e-04 1.000e+04 -1.000e+04
0.000e+00 1.000e-04
>>
```

⇒ If $b_1 \times$ [illegible] zero

⇒ Addition of floating [illegible] $+ c)$

*Assume $e_2 > e_1$

Floating point value representation

$$\pm b \times 2^{e}$$

- On most machines, **double**s are represented using 8 bytes (64 bits)
  - These are (roughly) used to encode:
    - The sign ($\pm$): 1 bit
    - The signed (integer) exponent (e): 11 bits
    - The unsigned (integer) base (b): 52 bits

Floating point value representation

- On most machines, ints are represented using 4 bytes (32 bits)
- On most machines, floats are represented using 4 bytes (32 bits)
  - In order to interpret the 4 bytes as a number, we need to know the type!

```
#include <stdio.h>
int main( void )
{
        float f = 1.f;
        float *f_ptr_as_float_ptr = &f;
        int *f_ptr_as_int_ptr = (int*)f_ptr_as_float_ptr;
        int i = *f_ptr_as_int_ptr;
        printf( "%.2f %d\n" , f , i );
        return 0;
}
```

```
>> ./a.out
1.00 1065353216
>>
```

Outline

- Floats
- From code to executables
  - Header files
  - Independent compilation and linking
  - `make` and `Makefiles`

Source code

Separate source files

- Big software projects are typically split among multiple files
- Code accomplishing related tasks is often grouped together (forming a library of functions)
- Different developers may create/edit/test different pieces

Header files

Q: How do different files in a software package communicate?

A: When compiling functions in one file, we need the declarations of functions in the other file

- ✗ We could include the declarations at the beginning of the file
  - This causes "code bloat" and makes it hard to see what the code is doing
- ✓ We gather declarations in header (`.h`) files and then `#include` the header files
  - A separate source (`.c`) file will contain definitions for functions declared in the header file
    - Typically, functions defined in `file-name.c` are declared in a function named `file-name.h`
  - We use `#include <file-name.h>` when the header file is part of the general library
  - We use `#include "file-name.h"` when the header file is ours

Header files

```c
#include "func.h"

float mult2add( int x , float y )
{
        return mult2( x ) + y;
}

int mult2( int a )
{
        return 2*a;
}
```

func.c

```c
float mult2add( int x , float y );
int mult2( int a );
```

func.h

```c
#include <stdio.h>
#include "func.h"

int main( void )
{
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c func.c
>> ./a.out
7.00    14
>>
```

Header files

Note:

If we do not include the .h file(s), the compiler can try to guess the declaration:

- It can try to guess the types of the function's input from the arguments passed
- It will assume the output is an int
  - This is right for mult2
  - This is wrong for mult2add

```
#include <stdio.h>
// #include "func.h"

int main( void )
{
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
float mult2add( int x , float y );
int mult2( int a );
```

func.h

Header files

Note:

If we do not include the .h file(s), the compiler can try to guess the declaration:

```
#include <stdio.h>
// #include "func.h"

int main( void )
{
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
```

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c func.c
mainFile.c: In function main:
mainFile.c:6:20: warning: implicit declaration of function mult2add [-Wimplicit-function-declaration]
 printf( "%.2f\t" , mult2add( 2 , 3.f ) );
                    ^~~~~
mainFile.c:6:13: warning: format %f expects argument of type double, but argument 2 has type int [-Wformat=]
 printf( "%.2f\t" , mult2add( 2 , 3.f ) );
             ^
mainFile.c:7:19: warning: implicit declaration of function mult2 [-Wimplicit-function-declaration]
  printf( "%d\n" , mult2( 7 ) );
                   ^~~~~
>>
```

func.h

Header files

Note:

If we do not include the .h file(s), the compiler can try to guess the declaration:

```
#include <stdio.h>
// #include "func.h"

int main( void )
{
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
```

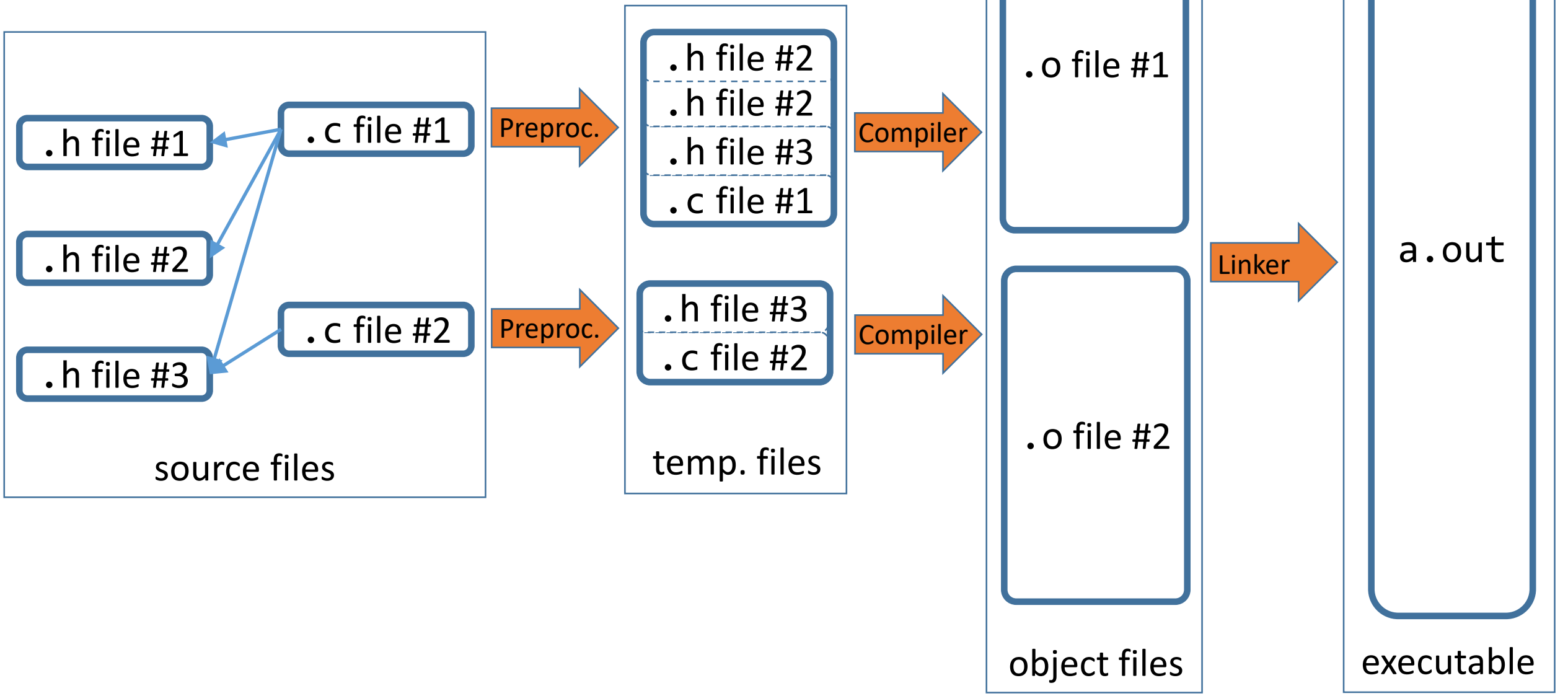
```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c func.c
mainFile.c: In function main:
mainFile.c:6:20: warning: implicit declaration of function mult2add [-Wimplicit-function-declaration]
 printf( "%.2f\t" , mult2add( 2 , 3.f ) );
                    ^~~~~
mainFile.c:6:13: warning: format %f expects argument of type double, but argument 2 has type int [-Wformat=]
 printf( "%.2f\t" , mult2add( 2 , 3.f ) );
             ^
mainFile.c:7:19: warning: implicit declaration of function mult2 [-Wimplicit-function-declaration]
  printf( "%d\n" , mult2( 7 ) );
                   ^~~~~
>> ./a.out
0.00    14
>>
```

The answer is junk because the compiler is reading the 4 float bytes as 4 int bytes

Compiling and linking

- Until now, we've used one `gcc` command for compilation and linking
  - *compiling* translates source (`.c`) files into intermediate object (`.o`) files
  - *linking* combines `.o` files into one executable file called `a.out`
    (Recall that we can optionally specify the executable name with the `-o` flag)

Compiling and linking



Using header files

When we've run gcc, we did all three steps at once:

- Pre-processing
- Compilation
- Linking

```
#include <stdio.h>
#include "func.h"

int main( void )
{

        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c func.c
>> ./a.out
7.00    14
>>
```

Using header files

When we've run gcc, we did all three steps at once:

- Pre-processing
- Compilation
- Linking

If we modify one of the files, we need to recompile

But we only need to generate new object files for the modified source files

```c
#include <stdio.h>
#include "func.h"

int main( void )
{
        printf( "hi\n" );
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c func.c
>> ./a.out
hi
7.00    14
>>
```

Using header files

When we've run gcc, we did all three steps at once:

- Pre-processing
- Compilation
- Linking

If we modify one of the files, we need to recompile.

But we only need to generate new object files for the modified source files

```
#include <stdio.h>
#include "func.h"

int main( void )
{
        printf( "hi\n" );
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
>> gcc -std=c99 -pedantic -Wall -Wextra -c main.c
>> gcc main.o func.o
>> ./a.out
7.00    14
>>
```

We can separately invoke the compiler (with preprocessor) and the linker

`make` and `Makefiles`

- Separately invoking the compiler and linker can be a pain:
  - We need to track dependencies
  - We need to track which files changed since the last time we compiled / linked

`make` and `Makefiles`

- `make` is a tool that helps keep track of which files need to be reprocessed so that those, and only those, are recompiled

- It takes a file containing a list of rules for generating specific files / targets:
  - *Prerequisites*: What targets does this target depend on?
  - *Recipes*: What should be done to generate this target?

`make` and `Makefiles`

- `make` is a tool that helps keep track of which files need to be reprocessed so that those, and only those, are recompiled

- It takes a file containing a list of rules for generating specific files / targets
  - Simplest to name the file `Makefile` or `makefile`, otherwise need to run the `make` command with extra flags (specifying the name of the configuration file)
  - There are strict rules about structure of `Makefile`, so it's easiest to follow a template and modify
  - Note that tabs and spaces are not equivalent in a `Makefile`!

Makefiles

Lines in a `makefile` consist of

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Makefiles

Lines in a `makefile` consist of:

- *Comments*, start with a # sign

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Makefiles

Lines in a `makefile` consist of:

- *Comments*, start with a # sign
- *Definitions,* assigned as:

`constant-name=...`

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Makefiles

Lines in a `makefile` consist of:

- *Comments*, start with a # sign
- *Definitions,* assigned as:

  `constant-name=...`

  and later referenced as:

  `$(constant-name)`

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Makefiles

Lines in a `makefile` consist of:

- *Comments*, start with a # sign
- *Definitions,* assigned as:

  `constant-name=...`

  and later referenced as:

  `$(constant-name)`
- *Rules* for generating the targets

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

`Makefile` rules

- Format of a `Makefile` rule
  - `target-name: {dependencies}`

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

`Makefile` rules

- Format of a `Makefile` rule
  - `target-name: {dependencies}`
  - a set of lines with a tab followed by a command-line instructions

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Invoking `Makefiles`

- Invoke the `main` tool with the name of the target to build
  - If no target is given, the first is used

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Invoking `Makefiles`

- Invoke the `main` tool with the name of the target to build
  - If no target is given, the first is used

Examples:

```
>> make clean
```

- Delete all object files and executable

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Invoking `Makefiles`

- Invoke the `main` tool with the name of the target to build
  - If no target is given, the first is used

Examples:

```
>> make func.o
```

- Has `func.c` or `func.h` changed since the last creation of `func.o`?
  - Yes: Compile `func.c` → `func.o`
  - No: Do nothing

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Invoking `Makefiles`

- Invoke the `main` tool with the name of the target to build
  - If no target is given, the first is used

Examples:

```
>> make main.o
```

- Has `main.c` or `func.h` changed since the last creation of `main.o`?
  - Yes: Compile `main.c` → `main.o`
  - No: Do nothing

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Invoking `Makefiles`

- Invoke the `main` tool with the name of the target to build
  - If no target is given, the first is used

Examples:

```
>> make
>> make main
```

- Does `main.o` or `func.o` need to be (re-)generated?
  - Yes: `make` those first
- Has `main.o` or `func.o` changed since the last creation of main?
  - Yes: Link `main.o` + `func.o` → `main`
  - No: Do nothing

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# Rule for clean-up
clean:
	rm -f *.o
	rm -f main
```

Invoking `Makefiles`

- Invoke the `main` tool with the name of the target to build
  - If no target is given, the first is used

Examples:

```
>> make
>> make main
```

- Does `main.o` or `func.o` need to be (re-)generated?
  - Yes: `make` those first

```
# Define the compiler and flags
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

# (Default) rule for making the main file
main: main.o func.o
	$(CC) -o main main.o func.o

# Rule for making the main object file
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# Rule for making the functions object file
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c
```

Note:
This means that `make` can have a cascading effect, with the `make`ing of one target requiring the `make`ing of another.

Piazza → Resources section → Resources tab → Exercise 3-2