Intermediate Programming

Day 8

Outline

- Exercise 7
- Separate compilations
- Makefiles
- Header guards
- Review questions

Exercise 7

- Declare the **div** function

functions.c

```
...
float div( float , float );

int main()
{
        ...
}
...
```

It’s bad coding style not to name the variables in the declaration.

Exercise 7

- Declare and define the **mult** function

functions.c

```
…
float mult( float , float );

int main()
{
        …
}

float mult( float a , float  b ){ return a*b; }
```

Exercise 7

- Declare and define the fac function

functions.c

```
…
int fac( int );

int main()
{
        …
}

…
int fac( int a )
{
        if( a<0 ) return 0;
        else if( a==0 ) return 1;
        else return a*fac(a-1);
}
```

Exercise 7

- Declare and define the **bsearch** function

functions.c

```
…
int bsearch( float [] , int , int , float );

int main()
{
        …
}

…
int bsearch( float ra[] , int low, int high , float target )
{
        if( low>high ) return -1;
        if( low==high ) return ra[low]==target ? low : -1;
        int mid = (low+high)/2;
        if( ra[mid]==target ) return mid;
        else if( ra[mid]>target ) return bsearch( ra , low , mid-1 , target );
        else                      return bsearch( ra , mid+1 , high , target );
}
```

Note the use of the ternary operator.

Exercise 7

- Declare and define the **bsearch2** function

functions.c

```
…
int bsearch2( float [] , int , int , float , float [] , int );

int main()
{
        …
}

…
int bsearch2( float ra[] , int low , int high , float target , float results[] , int size )
{
        if( low>high ) return -1;
        if( low==high )
        {
                results[size++] = low;
                return ra[low]==target ? low : -1;
        }
        int mid = (low+high)/2;
        results[size++] = mid;
        if( ra[mid]==target ) return mid;
        else if( ra[mid]>target ) return bsearch2( ra , low , mid-1 , target , results , size );
        else                      return bsearch2( ra , mid+1 , high , target , results , size );
        return size;
}
```

Outline

- Exercise 7
- Separate compilations
- Makefiles
- Header guards
- Review questions

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;
```

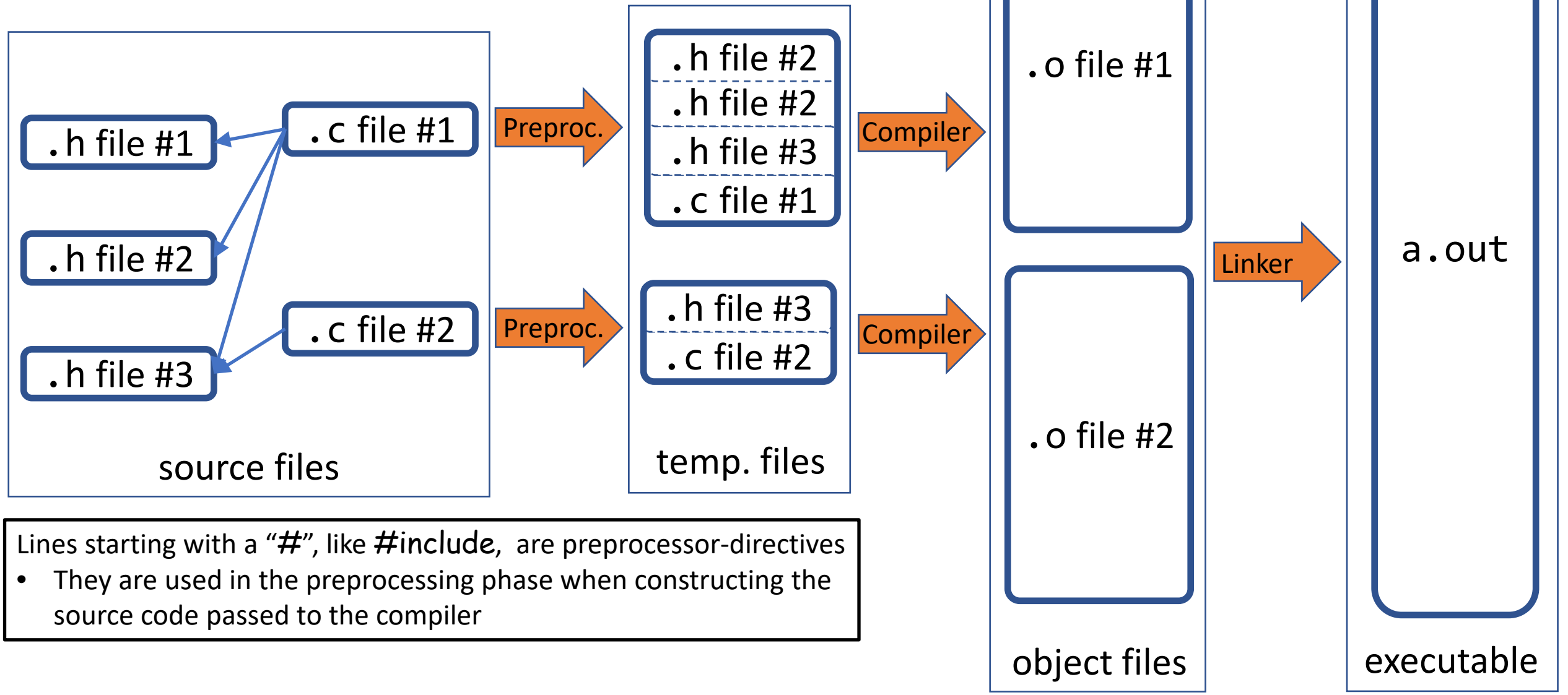
```
>> gcc -std=c99 -pedantic -Wall -Wextra main.c func.c
mainFile.c: In function main:
mainFile.c:6:20: warning: implicit declaration of function mult2add [-Wimplicit-function-declaration]
 printf( "%.2f\t" , mult2add( 2 , 3.f ) );
                    ^~~~~
mainFile.c:6:13: warning: format %f expects 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



Lines starting with a "#", like **#include**, are preprocessor-directives

- They are used in the preprocessing phase when constructing the source code passed to the compiler

Using header files

When we run gcc, we can do 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 run gcc, we can do 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 (.o) 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 run gcc, we can do 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
hi
7.00    14
>>
```

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

Outline

- Exercise 7
- Separate compilations
- Makefiles
- Header guards
- Review questions

`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
```

*The braces indicate an optional argument.

`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
```

- make `main.o` and `func.o`
- 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
```

- make `main.o` and `func.o`
- Has `main.o` or `func.o` changed since the last creation of main?

```
# 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 makeing of one target requiring the makeing of another.

Outline

- Exercise 7
- Separate compilations
- Makefiles
- Header guards
- Review questions

Header guards

- You should enclose your header file with a header guard.
  - Preprocessor commands (starts with “#”) that ensure that your functions are only declared once

```
#ifndef FUNC_H
#define FUNC_H
float mult2add( int x , float y );
int mult2( int a );
#endif // FUNC_H
```

func.h

Header guards

- You should enclose your header file with a header guard.
  - Preprocessor commands (starts with "#") that ensure that your functions are only declared once
    - If the pre-processor variable FUNC_H has not been defined

```
#ifndef FUNC_H
#define FUNC_H
float mult2add( int x , float y );
int mult2( int a );
#endif // FUNC_H
```

func.h

Header guards

- You should enclose your header file with a header guard.
  - Preprocessor commands (starts with "#") that ensure that your functions are only declared once
    - If the pre-processor variable FUNC_H has not been defined
      - Then define it and declare the functions

```
#ifndef FUNC_H
#define FUNC_H
float mult2add( int x , float y );
int mult2( int a );
#endif // FUNC_H
```

func.h

Header guards

- You should enclose your header file with a header guard.
  - Preprocessor commands (starts with "#") that ensure that your functions are only declared once
    - If the pre-processor variable FUNC_H has not been defined
      - Then define it and declare the functions
    - Otherwise, don't do anything

```
#ifndef FUNC_H
#define FUNC_H
float mult2add( int x , float y );
int mult2( int a );
#endif // FUNC_H
```

func.h

Header guards

Example:

- The first time we #include func.h, FUNC_H is undefined, so we define it and include the declarations.

```
#include <stdio.h>
#include "func.h"
#include "func.h"
int main( void )
{
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
#include "func.h"

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

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

func.c

```
#ifndef FUNC_H
#define FUNC_H
float mult2add( int x , float y );
int mult2( int a );
#endif // FUNC_H
```

func.h

Header guards

Example:

- The second time we #include func.h, FUNC_H is defined, so the declarations are ignored.

```
#include <stdio.h>
#include "func.h"
#include "func.h"
int main( void )
{
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
#include "func.h"

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

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

func.c

```
#ifndef FUNC_H
#define FUNC_H
float mult2add( int x , float y );
int mult2( int a );
#endif // FUNC_H
```

func.h

Header guards

Example:

- If we don't have a header guard, the compiler doesn't mind. Yet. (As we include more complex C constructs in header files, it will.)

```
#include <stdio.h>
#include "func.h"
#include "func.h"
int main( void )
{
        printf( "%.2f\t" , mult2add( 2 , 3.f ) );
        printf( "%d\n" , mult2( 7 ) );
        return 0;
}
```

main.c

```
#include "func.h"

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

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

func.c

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

func.h

Outline

- Exercise 7
- Separate compilations
- Makefiles
- Header guards
- Review questions

Review questions

1. Why do we need header guards?

To keep from including the same declaration multiple times.

Review questions

2. What is the difference between compiling and linking?

Compiling creates objects files.

Linking joins the object files into an executable.

Review questions

3. What compiler flag do we use to create object files and what extension do those files have?

We use the `-c` flag.

The generated files will have a `.o` extension.

Review questions

4. What is a `target` in a Makefile?

Something (e.g. an object file, an executable, or an operation) that we want to construct/perform.

Review questions

5. What are the advantages of using Makefiles?

Keeps us from having to track what needs to be re-generated when a file has been modified.

Exercise 8

- Website -> Course Materials -> Exercise 8