Writing and compiling larger programs

Lecture 04.02

Given perfectly valid program

```
float total = 0.0;
short tax_percent = 6;

float get_with_tax(float f) {
  float tax_rate = 1 + tax_percent / 100.0;
  total = total + (f * tax_rate);
  return total;
}

int main() {
  float val = 12.30;
  printf("With tax: %.2f\n",
                get_with_tax(val));
 }
 return 0;
}
```

Change the order: it does not compile

```
float total = 0.0;
short tax_percent = 6;

int main() {
  float val = 12.30;
  printf("With tax: %.2f\n",
                get_with_tax(val));
 }
 return 0;
}

float get_with_tax(float f) {
  float tax_rate = 1 + tax_percent / 100.0;
  total = total + (f * tax_rate);
  return total;
}
```

?

```
gcc totaller.c -o totaller && ./totaller
totaller.c: In function "main":
totaller.c:14: warning: format "%.2f" expects type
"double", but argument 2 has type "int“
totaller.c:23: error: conflicting types for “get_with_tax"
totaller.c:14: error: previous implicit declaration of
“get_with_tax" was here
```

The logic of GCC: 1

```
float total = 0.0;
short tax_percent = 6;

int main() {
  float val = 12.30;
  printf("With tax: %.2f\n",
              get_with_tax(val));
 }
 return 0;
}

float get_with_tax(float f) {
  float tax_rate = 1 + tax_percent / 100.0;
  total = total + (f * tax_rate);
  return total;
}
```

get_with_tax()
returns int

Here's a call to a function I've never heard of.
I'll keep a note of it for now and find out more later.
I bet the function **returns an *int***.
Most do.

The logic of GCC: 2

```
float total = 0.0;
short tax_percent = 6;

int main() {
  float val = 12.30;
  printf("With tax: %.2f\n",
                get_with_tax(val));
 }
 return 0;
}

float get_with_tax(float f) {
  float tax_rate = 1 + tax_percent / 100.0;
  total = total + (f * tax_rate);
  return total;
}
```

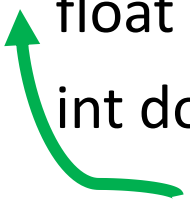
get_with_tax() returns int

A function called get_with_tax() that **returns a float**???
But in my notes it says we've already got one of these returning an int…


```
totaller.c:23: error: conflicting types for "get_with_tax"
totaller.c:14: error: previous implicit declaration of
"get_with_tax" was here
```

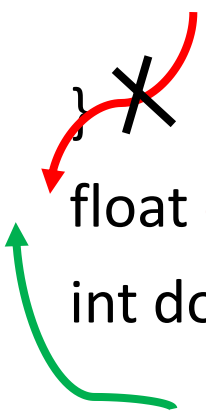
The order of functions matters to GCC

```
int do_whatever(){...}
float do_something_fantastic (int awesome_level) {...}
int do_stuff() {
        do_something_fantastic(11);
}
```



Keeping the order is painful

```
int do_whatever() {
        do_something_fantastic(5);
}
float do_something_fantastic (int awesome_level) {...}
int do_stuff() {
        do_something_fantastic(11);
}
```



And sometimes impossible

```
float ping() {
    ...
    pong();
    ...
}
```

```
float pong() {
    ...
    ping();
    ...
}
```

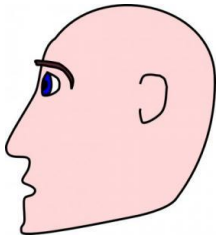
If you have two functions that call *each other*, then **one of them will always be called in the file before it's defined**

Solution: split the declaration and the definition

- Explicitly tell to the compiler what functions to expect
- When you tell the compiler about a function, it's called a function *declaration*:

float add_with_tax();

Function declaration does not have the body!



No assumptions – the code compiles

```
float total = 0.0;
short tax_percent = 6;
```

```
float get_with_tax(float f);
```

```
int main() {
  float val = 12.30;
  printf("With tax: %.2f\n",
                get_with_tax(val));
 }
 return 0;
}
```

```
float get_with_tax(float f) {
  float tax_rate = 1 + tax_percent / 100.0;
  total = total + (f * tax_rate);
  return total;
}
```

Declaration comes before use, and can be defined anywhere in the file

Put declarations into a header file

- The declaration is just a function ***signature***: name, parameters, and the type of return
- Once you've declared a function, the order of function definitions is not important
- **But even better**: take the whole set of declarations out and put them in a *header file*

Header files. Include

- Create a new file totaller.h:

float get_with_tax(float f);

- Include your header file in your main program

#include <stdio.h>

#include "totaller.h"

...

- When the preprocessor sees the #include in the code, it **copies its text into the source file**

Breaking code into multiple files: motivation

- Small programs -> single file

- “Not so small” programs :
  - Many lines of code
  - Multiple reusable components
  - More than one programmer

Example: Game code in a single file

game.c

```
#include <stdio.h>
int score = 0; // global variable
void update_score(int amt) {
...
}
void render_score() {
...
}
void render_board() {
...
}
void create_board(char *config) {
...
}
char *get_winner() {
...
}
void check_if_done() {
...
}
```

game.c (cont.)

```
int add_user(char *name) {
...
}
int remove_user(char *name) {
...
}
char *move_user(char *name) {
...
}
void end_game() {
...
}
void start_game() {
...
}
void reset_game() {
...
}
int change_level(int level_id) {
...
}
```

Regroup functions according to their logic

- The game functions could be regrouped into separate files, with each file containing a subset of functions dealing with a particular aspect of the game.
- There are many ways to divide these functions. One possible division may be the following:
  - Rendering functions (i.e. visual appearance)
  - Functions related to score-keeping
  - Functions which affect the game's state
  - Functions for maintaining user status

Functions distributed into multiple files

render.c	**score.c**	**state.c**	**users.c**

`render_score()`
`render_board()`
`update_score()`
	`check_if_done()`
	`get_winner()`
		`start_game()`
		`end_game()`
		`reset_game()`
		`change_level()`
			`add_user()`
			`remove_user()`
			`move_user()`

- To allow our program to make use of functions across various files, we need to add a header file with function declarations
- When parsing the code for compilation, GCC will verify the correct use of types and will link noted functions once it encounters their implementations

Header file: game.h

```
void update_score(int);
void render_score();
void render_board();
void create_board(char *);
char *get_winner();
void check_if_done();
int add_user(char *);
int remove_user(char *);
char *move_user(char *);
void end_game();
void start_game();
void reset_game();
int change_level(int);
```

- The file extension for header files is ".h", not ".c".
- You must specify the return type and parameter types for each function.
- You do not have to include the parameter names, but you're free to do so.

Include game.h into each c file

render.c	score.c	state.c	users.c

`#include "game.h"`
`render_score()`
`render_board()`
`update_score()`
	`#include "game.h"`
	`check_if_done()`
	`get_winner()`
		`#include "game.h"`
		`start_game()`
		`end_game()`
		`reset_game()`
		`change_level()`
			`#include "game.h"`
			`add_user()`
			`remove_user()`
			`move_user()`

- Problems:
- If a function in a header file is defined more than once across the various C files
- If a function in a header file is called, but not defined in any of the C files
- If the header file is included more than once

Include game.h into each c file

render.c	score.c	state.c	users.c

`#include "game.h"`
`render_score()`
`render_board()`
`update_score()`
	`#include "game.h"`
	`check_if_done()`
	`get_winner()`
		`#include "game.h"`
		`start_game()`
		`end_game()`
		`reset_game()`
		`change_level()`
			`#include "game.h"`
			`add_user()`
			`remove_user()`
			`move_user()`

- Problems:
- If a function in a header file is defined more than once across the various C files
- If a function in a header file is called, but not defined in any of the C files
- If the header file is included more than once

Include guards

- It is possible that different parts of the application ask for the same header file to be included
- To prevent compiler complaints about double declarations, you need to put include guards around the content of each header file, like this:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H
    Your declarations here
    and at the end of the file is:
#endif
```

Include guards

```
#ifndef HEADERFILE_H
#define HEADERFILE_H
    Your declarations here
    and at the end of the file is:
#endif
```

- Once the include “headerfile.h” is encountered, GCC checks if a unique value (in this case HEADERFILE_H) is defined
- Then if it's not defined, it defines it and continues to including the rest of the file
- When the the include “headerfile.h” is encountered again, the first ifndef fails, resulting in a blank file
- That prevents double declarations.

To fully understand how it works, we need to look at…

Four steps of compilation

1

Preprocessing: fix the source

Adds any extra header files it's been told about using the #include directive. Expands or skips over some sections of the program.

2

Compilation:

translate into assembly

Converts the C source code into assembly language: converts an if statement or a function call into a sequence of assembly language instructions.

```
movq -24(%rbp), %rax
movzbl (%rax), %eax
movl %eax, %edx
```

3

Assembly:

generate the object code

Assembles the symbol codes into *machine* or **object code**. This is the actual binary code that will be executed by the circuits inside the CPU. If you give the computer several files to compile for a program, it will generate a piece of object code for each source file.

4

Linking: put it all together

Fits pieces of object code together to form the **executable program**. The compiler will connect the code in one piece of object code that calls a function in another piece of object code

Sharing code - through linking

- Having game.h included in main.c will mean the compiler will know enough about, say, start_game() function to compile main.c into main.o (step 3)
- At the linking stage (step 4), the compiler will be able to connect the call to start_game() in state.c to the actual start_game() function implemented there
- To do all the four steps and compile everything together you just need to pass all the source files to GCC:

```
gcc score.c state.c render.c main.c -o game
```

Sharing variables

- Source code files normally contain their own separate variables
- If you want to share variables, you should declare them in your header file and prefix them with the keyword *extern*:

```
extern int passcode;
```

Summary: sharing code

- You can modularize code by dividing it between multiple C files
- Put the function declarations in a separate .h header file
- Include the header file in every C file that needs to use the shared code
- List all of the C files needed in the compiler command

Skipping some compilation steps

- If you've just made a change to one or two of your source code files, it's a waste to recompile every source file for your program.
- The compiler will run the preprocessor, compiler, and assembler for each source code file. Even the ones that haven't changed.
- And if the source code hasn't changed, the object code that's already generated for that file won't change either – and steps 1,2,3 for such files can be avoided

Compile the source into object files

- If you tell the compiler to save the object code into a file, it shouldn't need to recreate it unless the source code changes.
- If a file does change, you can recreate the object code for that one file and then pass the whole set of object files to the compiler so they can be linked.

```
gcc -c *.c
```

This will create object code for every c file.
Option -c tells the compiler that you want to create an object file for each source file, but you don't want to link them together into a full executable program

Create executable by linking object files

- Now that you have a set of object files, you can link them together with a simple compile command
- But instead of giving the compiler the names of the C source files, you tell it the names of the object files:

```
gcc *.o -o game
```

Recompile only file that changed

- Now you have a compiled program, just like before.
- But you also have a set of object files that are ready to be linked together if you need them again
- If you change just one of the files, you'll only need to recompile that single file and then relink the program:

```
gcc -c score.c

gcc *.o -o game
```

Simple rule for recompiling specific files

- How do you know if the score.o file needs to be recompiled from score.c?
- You just look at the timestamps of the two files.
  - If the score.o file is older than the score.c file, then the score.o file needs to be recreated
  - Otherwise, it's up to date

- If we have a simple rule, we can automate this process

Automate compilation with *make*

- The *make* tool will check the timestamps of the source files and the generated files, and then it will only recompile the files if things are out of date

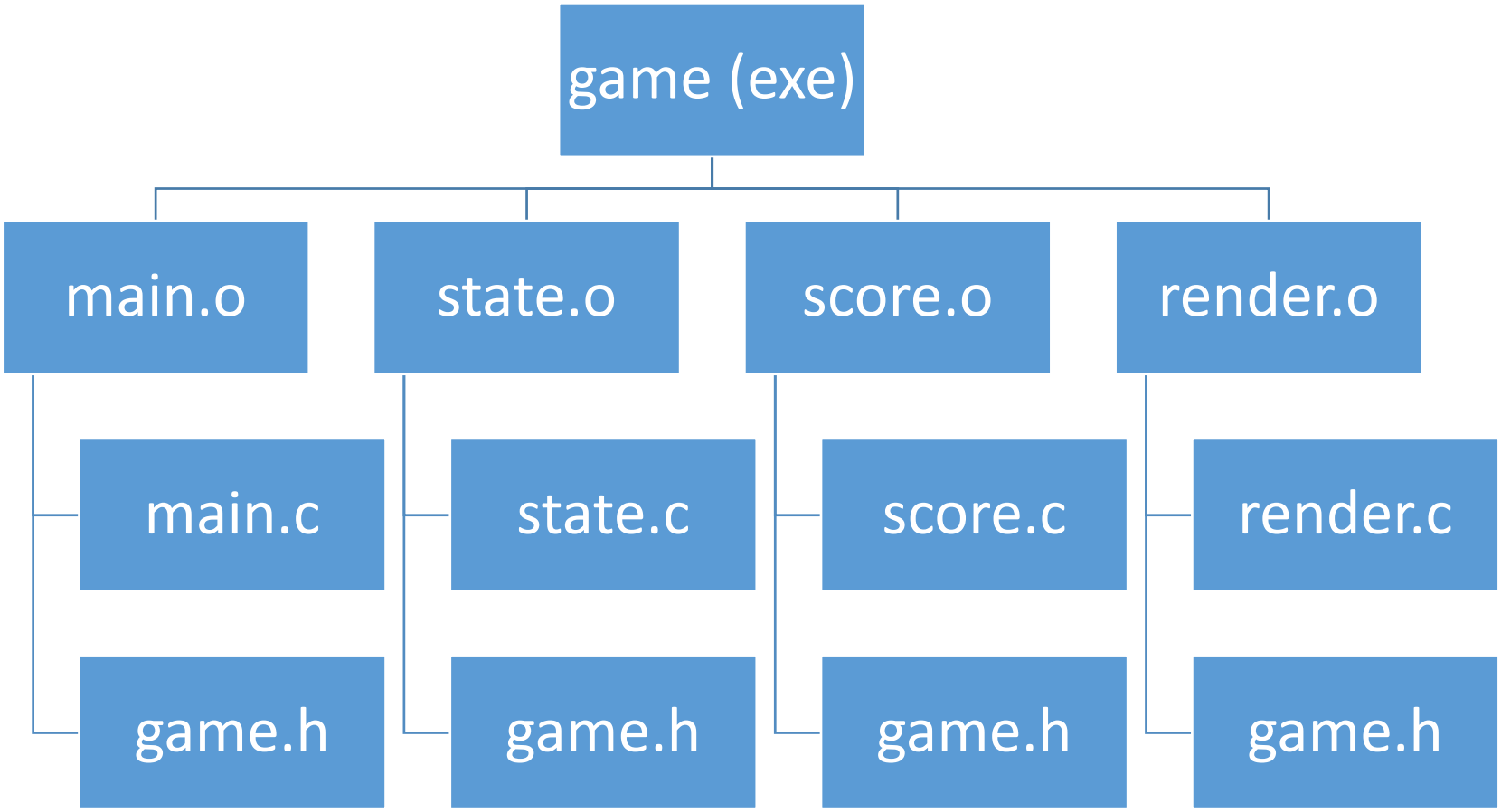
- Every file that *make* compiles is called a *target*
- For every target, *make* needs two things:
  - the dependencies - which files the target is going to be generated from
  - the recipe– the set of instructions it needs to run to generate the file

To write *makefile* we need to understand project structure

- Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- Example :
  - Program contains 5 files:

  main.c., state.c, score.c, render.c, and *game.h*
  - game.h is included in all .c files
  - The final executable should be called *game*

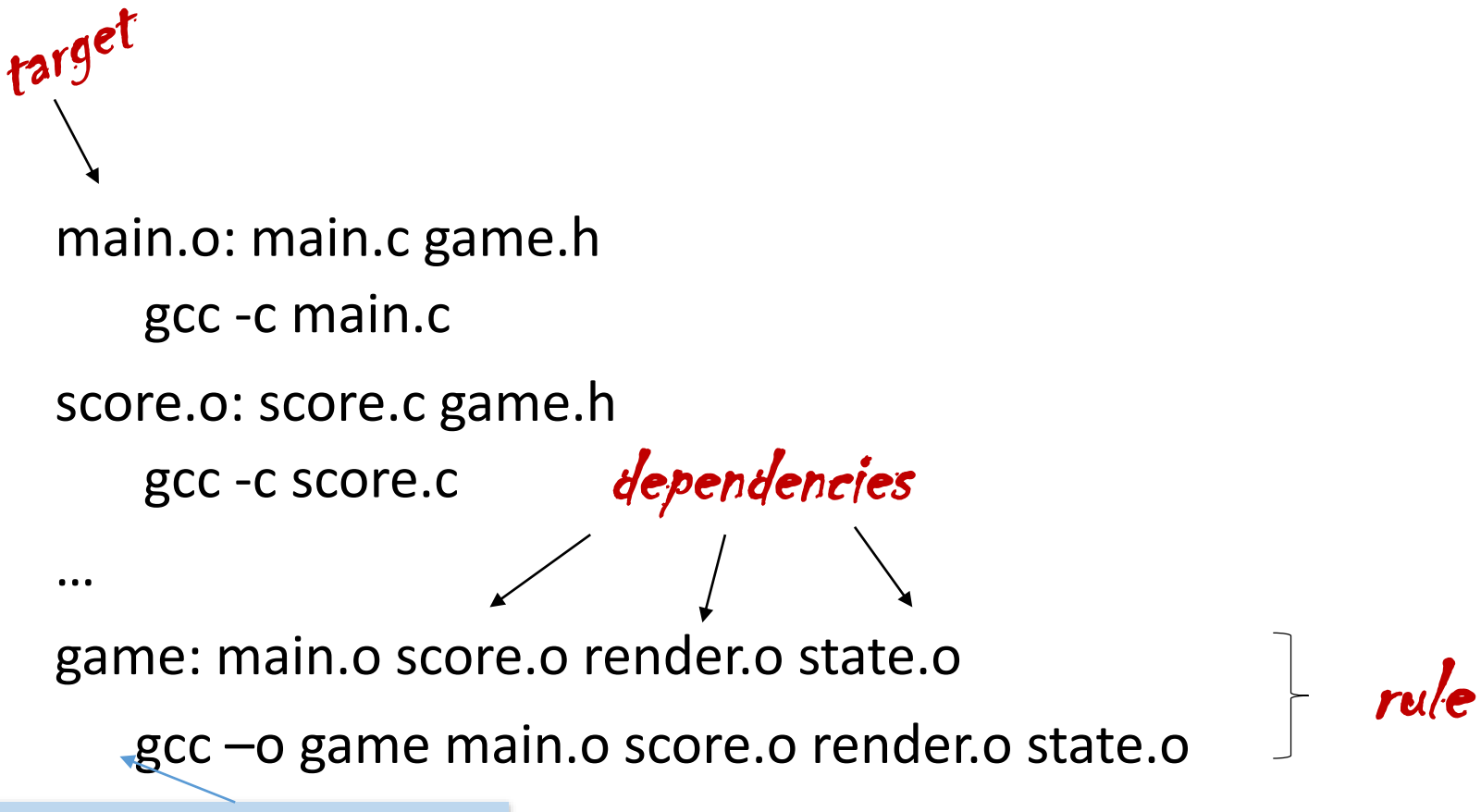
Sample project structure



Sample make file

```
main.o: main.c game.h
	gcc -c main.c
score.o: score.c game.h
	gcc -c score.c
render.o: render.c game.h
	gcc -c render.c
state.o: state.c game.h
	gcc -c state.c
game: main.o score.o render.o state.o
	gcc –o game main.o score.o render.o state.o
```

Sample make file



Shorter make file

- .o depends (by default) on corresponding .c file. Therefore, equivalent makefile is:

```
main.o: game.h
	gcc -c main.c
score.o: game.h
	gcc -c score.c
…
game: main.o score.o render.o state.o
	gcc –o game main.o score.o render.o state.o
```

How make operates

- Project dependencies tree is constructed
- Target of first rule should be created
- We go down the tree to see if there is a target that should be recreated. This is required when the target file is older than one of its dependencies
- In this case we recreate the target file according to the action specified, on our way up the tree. Consequently, more files may need to be recreated
- If something was changed, linking is performed

Minimum compilation

- *make* operation ensures minimum compilation, when the project structure is written properly
- Do not write something like:

```
game: main.c score.c state.c
  gcc -o game main.c score.c state.c
```

- This rule requires compilation of all project when something has changed

Minimum compilation: example

File	Last Modified
game	10:03
state.o	09:56
render.o	09:35
state.c	10:45
render.c	09:14
game.h	08:39

What should be recompiled?

Minimum compilation: example

File	Last Modified
game	10:03
state.o	09:56
render.o	09:35
state.c	10:45
render.c	09:14
game.h	08:39

state.o should be recompiled (state.c is newer)

Consequently, state.o is newer than game and therefore executable game should be recreated (by re-linking).

Using make

- Save your *make* rules into a text file called Makefile in the same directory
- Then, open up a console and type:

```
make game
```

Multiple targets

- We can define multiple targets for multiple executables in the same makefile
- Target *clean* – has an empty set of dependencies. Used to clean intermediate files.

```
make
```

- Will create all the executables

```
make clean
```

- Will remove intermediate files

Simple make tutorial

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/