Writing and compiling larger programs

Lecture 04.02

Given perfectly valid program

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

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

int main() {
  float val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
        printf("Total so far: %.2f\n",
                          add_with_tax(val));
        printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  return 0;
}
```

Change the order: it does not compile

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

int main() {
  float val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
        printf("Total so far: %.2f\n",
                        add_with_tax(val));
        printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  return 0;
}

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

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

The logic of GCC

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

int main() {
  float val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
        printf("Total so far: %.2f\n",
                          add_with_tax(val));
        printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  return 0;
}

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

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

Change the order: it does not compile

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

int main() {
  float val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
        printf("Total so far: %.2f\n",
                          add_with_tax(val));
        printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  return 0;
}

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

add_with_tax()
returns int

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

GCC

The order of functions matters to GCC

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

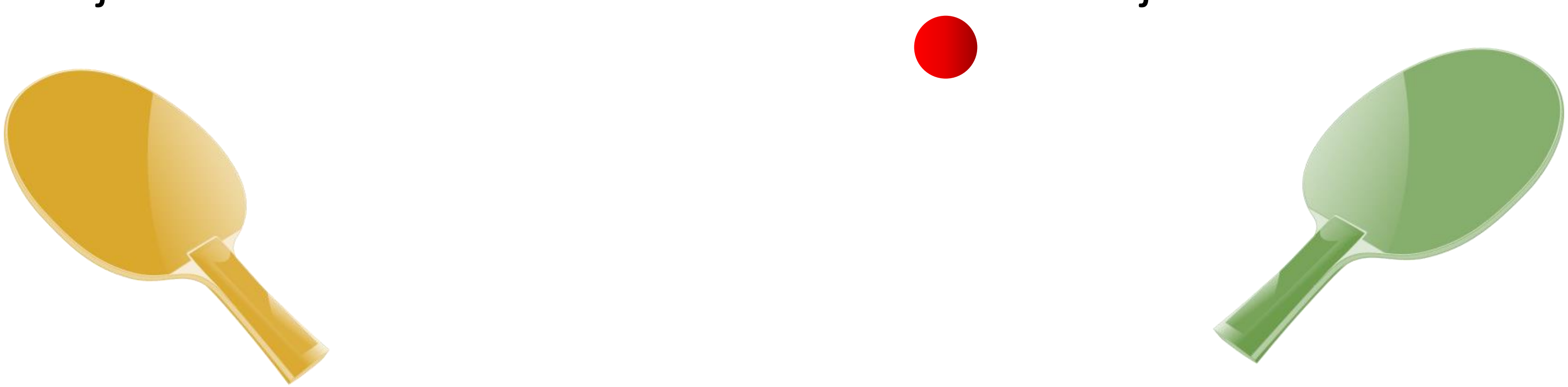
Keeping the order is painful

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

And sometimes impossible

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

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

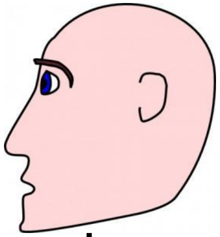


If you have two functions that call *each other*, then **one of them will always be called in the file before it's defined**

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 add_with_tax(float f);

int main() {
  float val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
        printf("Total so far: %.2f\n",
                              add_with_tax(val));
        printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  return 0;
}

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

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 that whole set of declarations out of your code and put them in a *header file*

Header files. Include

- Create a new file totaller.h:

float add_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
- 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 functions among different files

- Example: 2 specs

file_hider

Read the contents of a file and create an encrypted version using XOR encryption.

message_hider

Read a series of strings from the Standard Input and display an encrypted version on the Standard Output using XOR encryption

```
void encrypt(char *message)
```

XOR encryption

- Very simple way of disguising a piece of text by XOR-ing each character with some value
- The same code that can encrypt text can also be used to decrypt it.

```
void encrypt(char *message) {
    char c;
    while (*message) {
        *message = *message ^ 31;
        message++;
    }
}
```

0	0	0
0	1	1
1	0	1
1	1	0

0	0	0
0	1	1
1	0	1
1	1	0

Share functions through header

- If you are going to share the *encrypt.c* code between programs, you need some way to tell those programs about it

- You do that with a header file encrypt.h:

```
void encrypt(char *message);
```

- Include encrypt.h in both programs

Sharing code through linking

- Having encrypt.h inside the main program will mean the compiler will know enough about the encrypt() function to compile the code
- At the linking stage, the compiler will be able to connect the call to encrypt(msg) in message_hider.c to the actual encrypt() function declared in encrypt.h.
- Finally, to compile everything together you just need to pass the source files to gcc:

```
gcc message_hider.c encrypt.c -o message_hider
```

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 generated for that file won't change either.
- So if the compiler is generating the object code for every file, every time, what do you need to do?

Save object code into a file

- 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.

Compile the source into object files

gcc -c *.c

- This will create object code for every 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 launch
```

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 thruster.c
gcc *.o -o launch
```

Simple rule for recompiling specific files

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

Automate this process 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

Sample make file

target   dependencies

```
launch.o: launch.c launch.h thruster.h
	gcc -c launch.c
thruster.o: thruster.h thruster.c
	gcc -c thruster.c
launch: launch.o thruster.o
	gcc launch.o thruster.o -o launch
```

rule

The recipe must begin with a tab character

Using make

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

Make launch

Make has to work on teaching lab machines!

Q: If I write a Makefile for a Windows machine, will it work on a Mac? Or a Linux machine?

A: Because makefiles calls commands in the underlying operating system, sometimes makefiles don’t work on different operating systems.

Example: make with macros and variables

Target: dependencies

$@ : $^

```
CC = gcc
CFLAGS = -O3 -Wall
CFLAGS += -D_LARGEFILE_SOURCE
CFLAGS += -finline-functions
CFLAGS += -funroll-loops
MATHFLAG=-lm

# Source files
SC_SRC=common.c dna_common.c keyword_tree.c kmers_to_kwtree.c count_kmers.c streamcount.c

# Targets
all: streamcount

#streams the lines of the input file and counts k-mers
streamcount: $(SC_SRC)
	$(CC) $(CFLAGS) $^ -o $@ ${MATHFLAG}
clean:
	rm streamcount
```

Full target name for all

Simple make tutorial

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