601.220 Intermediate Programming

Spring 2023, Day 8 (February 8th)

Today's agenda

- Exercise 7 review
- Separate compilation, Makefiles, header guards
- Exercise 8

Reminders

- HW1 due Friday
- HW2: first "written" homework, to be released Friday, due next Friday (Feb 17th)
  - Note that late days may not be used on written assignments

Exercise 7 review

Adding a function declaration (a.k.a. "function prototype") for the `div` function:

```
float div(float a, float b);
```

A function declaration makes the compiler aware of the name, parameter type(s), and return type of a function so that calls to the function can be checked for correct usage.

Exercise 7 review

`mult` function declaration:

```
float mult(float a, float b);
```

`mult` function definition:

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

Exercise 7 review

$n! = 1 \times 2 \times \ldots \times (n-1) \times n$

$(n-1)!$

`fac` declaration:

```
long fac(int a);
```

`fac` definition (observations: $0! = 1$, $n! = (n-1)! \times n$ when $n > 0$):

```
// Precondition: a >= 0
long fac(int a) {
  assert(a >= 0);
  if (a == 0) { return 1; }
  return fac(a - 1) * a;
}
```

if (a < 0) return 0;

Exercise 7 review

bsearch function:

```
int bsearch(float ra[], int low, int high, float target) {
  // base cases
  if (low > high)  { return -1; }
  if (low == high) { return (ra[low] == target) ? low : -1; }
  int mid = low + ((high-low)+1) / 2;
  if (ra[mid] == target) { return mid; }
  // ...recursive cases left as exercise for reader...
}
```

Exercise 7 review

`bsearch2`: The caller of `bsearch2` can't know how many values were added to the `results` array because the `size` parameter is passed by value.

Day 8 recap questions

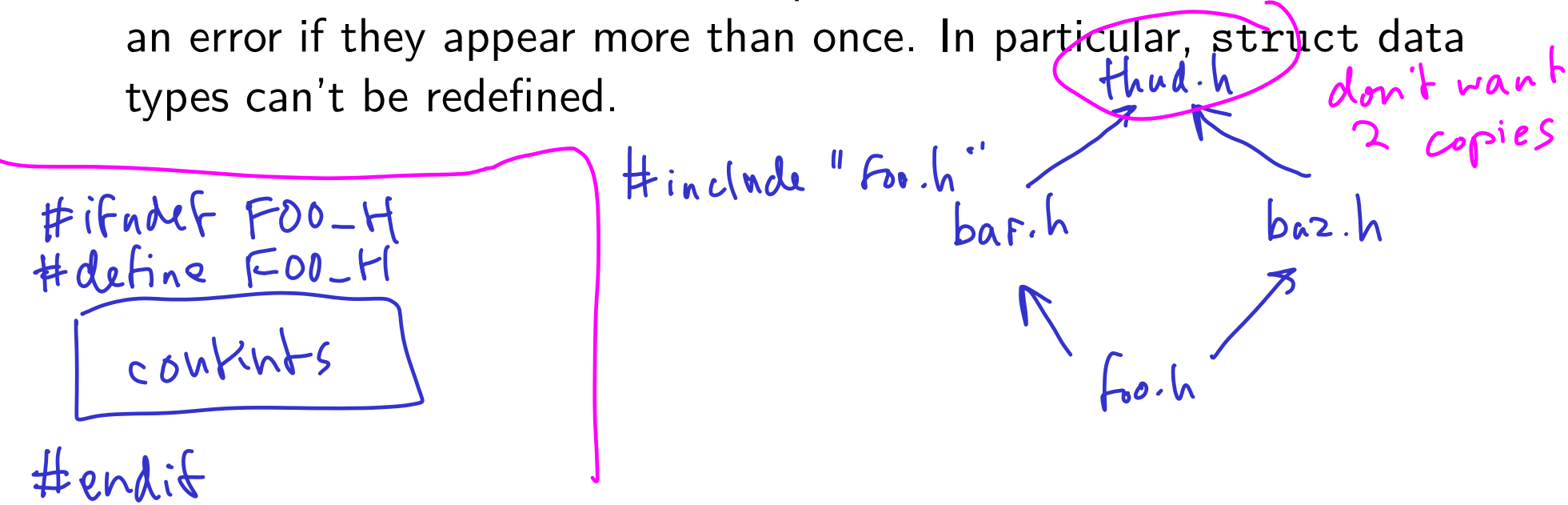
1. Why do we need header guards?
2. What is the difference between compiling and linking?
3. What compiler flag do we use to create object files and what extension do those files have?
4. What is a `target` in a Makefile?
5. What are the advantages of using Makefiles?

1. Why do we need header guards?

The `#include` directive means (essentially) "copy the contents of the named header file into the source code module that is being compiled."

Header guards prevent the contents of a header file from being copied more than once.

Some constructs which often are placed in a header file could cause an error if they appear more than once. In particular, `struct` data types can't be redefined.



2. What is the difference between compiling and linking?

Compiling: converts C source code (`.c` source file) into "object code" (`.o` "object file").

Linking: combines one or more `.o` (object) files into an executable file. Also, resolves calls to library functions (such as `printf`, `scanf`, `pow`, etc.)

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

The `-c` option means "compile source code to an object file".

Object files have a `.o` file extension.

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

A Makefile target is a file to be created based on the contents of other files.

For a C program, typically the Makefile will have targets for each object (`.o`) file, as well as a target for the executable.

The first target in a Makefile is the *default* target. If `make` is invoked without specifying an explicit target to build, the default target is built.

5. What are the advantages of using Makefiles?

Makefiles *automate* the process of compiling and linking a program. Just run the command `make`, and (if the Makefie is written correctly) the entire program will be compiled and linked.

If each target properly lists its *dependencies*, then `make` will figure out exactly which compiler commands need to be run, and in what order. In general, if any targets are "out of date", meaning that one or more dependencies is newer, or has been modified since the target was built, `make` will know to rebuild the target.

TL;DR when you have a properly-written Makefile all you need to do to build the program is run the command `make`.

Makefile tips

Definitions (at top of the `Makefile`):

```
CC = gcc
CFLAGS = -std=c99 -Wall -Wextra -pedantic
```

Target for an executable:

```
fooprog : foo.o bar.o
	$(CC) -o fooprog foo.o bar.o
```

Target for an object file:

```
foo.o : foo.c header1.h header2.h
	$(CC) $(CFLAGS) -c foo.c
```

Makefile syntax

Note that the *commands* for a target *must* begin with a tab character.

`emacs` and `vim` will show you if any comands aren't indented with a tab character.

Exercise 8

- Implement the `concat` string-concatenation function
- Split `all_in_one.c` into separate source files as follows:
  - `run_concat.c` should contain the `main` function
  - `string_functions.h` should have a declaration of `concat`
  - `string_functions.c` should have the definition of `concat`
- Modify the `Makefile`:
  - it should have targets for `run_concat` (the executable), `run_concat.o`, and `string_functions.o`
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes