make and Makefiles

- `make` is a tool that helps you keep track of which files need to be recompiled
  - Save time by not re-compiling unnecessarily
  - Avoid headaches from forgetting to recompile code that you changed!
  - Save yourself lots of typing
- in a configuration-type file called `Makefile`, carefully specify which files depend on which other files, and which commands should be used to create them

make and Makefiles

- Simplest to name the file `Makefile` or `makefile`, otherwise need to run `make` command with extra flags
- There are very strict rules about structure of Makefile, so easiest to follow a template and modify
- Beware: tabs and spaces are not equivalent in a Makefile!

make and Makefiles

- Lines in a makefile that begin with `#` are comments
- May define symbolic constants using `$` operator, e.g. `CFLAGS=-std=c99 -pedantic -Wall -Wextra`, then refer to them in a command using `$(constant-name)`, e.g. `$(CFLAGS)`
- Then list any number of rules...
  - First (topmost) target listed is default target to run
- Format of a Makefile rule
  - `target_name:` list of files on which target depends
  - TAB followed by command-line instruction to generate target
- Multiple targets can be triggered by making a single target
  - If you make target `main`, then first any files on which `main` depends will be re-made if not up-to-date

make and Makefiles

```
// Makefile:
CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra

main: mainFile.o functions.o
	$(CC) -o main mainFile.o functions.o

mainFile.o: mainFile.c functions.h
	$(CC) $(CFLAGS) -c mainFile.c

functions.o: functions.c functions.h
	$(CC) $(CFLAGS) -c functions.c

clean:
	rm -f *.o main
```

symbolic constant

target named main

tab character

tab character

tab character

dependencies of the target main: it means in order to be able to run target main, which is "gcc -o main mainFile.o functions.o", mainFile.o and functions.o should exist and be up-to-date

removes all .o files as well as the executable "main". Useful to trigger a fresh compilation of all .c files and produce a fresh executable

```
$ make clean
rm -f *.o main
$ make
gcc -std=c99 -pedantic -Wall -Wextra -c mainFile.c
gcc -std=c99 -pedantic -Wall -Wextra -c functions.c
gcc -o main mainFile.o functions.o
$ ./main
5.00 14
```

Using make: commands to type at command prompt

- `make functions.o`
  - compiles (or re-compiles) functions.c if needed, to create `functions.o`
  - re-compiling is needed if *either* `functions.c` or `functions.h` has changed, since the `functions.o` target lists both files in its dependency list
- `make mainFile.o`
  - compiles (or re-compiles, if needed) mainFile.c if needed, to create `mainFile.o`
  - re-compiling is needed if *either* `mainFile.c` or `functions.h` has changed, since the `functions.o` target lists both files in its dependency list
- The above commands are helpful, but aren't usually what we need...

Using make: commands to type at command prompt

- `make main`
  - links (or re-links, if needed) `mainFile.o` and `functions.o` to create an executable we decided to call `main` (see the `-o` flag?)
  - first it checks that `mainFile.o` and `functions.o` are up-to-date, based on the target rules specified for these (so `make` can have a cascading effect through multiple rules)
  - there's nothing special about the name `main` as the target here; we could've called this target `bob` if we'd wanted
- `make`
  - has same effect as `make main`, since `main` was listed as first target in `Makefile`
  - this is what we'll type most often; it's the quickest way to get the entire program built!
- `make clean`
  - removes intermediate files and executable called `main`, so we can start fresh