Structuring Projects

- It is infeasible to try to write any useful system as a single source file
  - Especially if we'd like to reuse parts of it in other projects.
  - Or if we'd like someone else to clearly understand its structure.
- So today we will look at how we can structure our source into a project, splitting it into multiple files and possibly libraries.

Object File Compilation and Linking

- So far, we have allowed the compiler to hide from us a crucial step in building software: the **linking step**.
- When we run `gcc hello_world.c -o hello_world`, the compiler first **compiles** all of the files (in this case `hello_world.c`) to *object* files (the actual machine code representing each C file)
- Then it automatically **links** them together into a single file (the final executable), such that references to functions and variables foreign to each object file are resolved as calls (*i.e.* jumps to addresses) into the other object files.
- Additional to our object files, compiled code from the standard libraries is also pulled into the final executable (*e.g.* the machine code of, say, `printf`).
    - In fact, for a small program, most of the code in the executable will be code from such libraries.

Object File Compilation and Linking

- Often we require explicit control over the build process, so we first build the object files for each C file, using the *compile-only* flag, `-c`:
  - `gcc -c hello_world.c -o hello_world.o`
- Then we link the object files into the final executable (in this case `my_app`):
  - `gcc -o my_app hello_world.o linked_list.o ...`
- For the interested, if you run `objdump -d <some_compiled_binary>` on some object file or executable you will see a list of functions defined within the file along with their disassembled code.

Today's Code

- Today we will look at the code of a simple application, `todo_list`, that allows the user to enter a list of items, such that we will get a better understanding of what is involved in putting together a larger project.
  - First we will see how we could implement it as multiple files, looking at how header files my be defined and used.
  - Then we will see how we could split off the linked list code into to a library which can be re-used in other applications.

Building a Project with Make

- We can use the tool `make` to help us to build large software projects.
- `make` allows us to automate the running of long commands, through the definition rules in a special file called `Makefile`.

Structure of simple Makefile :

- Starts with declarations (assignment of values to variables)
- Then have list of targets and commands which re-create the target
  Need to have TAB-character at beginning of line containing the command!
- Can call `make` in directory `<dir>` via
  `make -C <dir>`

- Such a list of targets consists of one line containing target and dependencies
- dependencies are files which need to be present and newer than the target
- commands will be executed if target needs to be re-generated variables may be used

Conventions:

- Have target `all` which makes everything in this directory normally first target
- Have also target `clean` which removes all targets and temporary files created