General

Opening tarball: `tar xzf filename.tgz`
Compiling C: `gcc -Wall -g filename.c -o filename`

I strongly recommend telling `gcc` to warn about empty parameter lists by adding `-Wstrict-prototypes` to the compilation flags (along with `-Wall`)

- The `-Wall` option to gcc enables many warnings - I strongly recommend that you always use it.
- The `-g` option to gcc includes debugging information in the generated executable file (useful when debugging with `gdb`).

Run executable: `./filename`

By default, gcc accepts a GNU dialect broadly equivalent to C11. Fix which standard to use by passing `-std=c89` or `-std=c99` to gcc.
C99 introduces a `bool` type, defined in `<stdbool.h>`.

- In C99, single line comments beginning with `//` were introduced. Multi-line comments (`/* ... */`) are allowed in all C dialects.

`file` tries to identify what kind of file it's given: `file filename.extension`

- nm lists symbols ("names") inside an object file - and whether they are defined in that object file, or undefined (used by that object file, but defined elsewhere). For example `nm welcome.o` reports:

```
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U puts
```

  Showing that it defines main and has undefined references to puts and whatever that offset table symbol may be. These will be found in the C library during linking.
- To look inside the C library, we have to find it. `find /usr/lib -name libc.a` reports:

```
/usr/lib/x86_64-linux-gnu/libc.a
...
```

- `ar t /usr/lib/x86_64-linux-gnu/libc.a` shows that libc contains hundreds of object files.
- `nm /usr/lib/x86_64-linux-gnu/libc.a` lists all the symbols inside all those object files. Is puts defined in libc?

  `nm /usr/lib/x86_64-linux-gnu/libc.a 2>/dev/null | grep puts` finds:

  ```
  ioputs.o:
  0000000000000000 T _IO_puts
  0000000000000000 t _IO_puts.cold
  0000000000000000 W puts
  ```

  So, yes, puts is defined in libc, inside an object file ioputs.o.
- `nm` may also be run on `welcome` and surprisingly, it still shows undefined symbols. Why?
- `file welcome` reports:

  ```
  welcome: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked...
  ```
- `ldd welcome` will show you any dynamically linked libraries:

  ```
  linux-vdso.so.1 (0x00007fffb6d9a000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f55638b4000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f5563af6000)
  ```

  showing us that the C library is actually being dynamically linked (hence why there are undefined symbols in `welcome`).

Look at `make` in C Tools

CBuild

- The traditional answer is a tool called make, which I'll describe in the first of my C Tools lectures. I also present a simpler tool of mine called CBuild. Here in these lectures, let's use CBuild's cb and cr commands. (They are already installed on lab computers, and you can install them yourself anywhere else by following the instructions in the README from the first C Tools tarball.)
- Using cb to compile our welcome.c example onto welcome, all we need do is type:

```
cb welcome
```

  and cb will figure everything out, realising that welcome.c is the source of welcome, and then taking the minimum number of actions necessary to rebuild welcome.
- If you want to force a full recompilation, simply say:

```
cb --allclean welcome
```

To compile using the CBuild file, type `cb`
To remove all compiled files, `cb --clean`
To remove all compiled files, and then recompile, `cb --allclean`

Use CBuild!

- If you want to tell cb what executables to build, what gcc flags to use, or what additional libraries to link in, you can prepare a .cbuild file, for example:

```
BUILD = welcome program2 program3
CFLAGS = -Wall -g
LDLIBS = -lm
```

- After doing that, cb doesn't need you to tell it what to build, so:

```
cb
```

  will attempt to compile and link all 3 programs, and

```
cb --clean
```

  will clean all the build artefacts (object files and executables from all 3 programs).

- As a convenience, I have recently added a companion tool called cr - to compile and run welcome.c, just run:

```
cr welcome.c
```

- Of course, cr will not attempt to run the executable if compilation or linking failed.
- See man cb, and my first C Tools lecture, for more details.