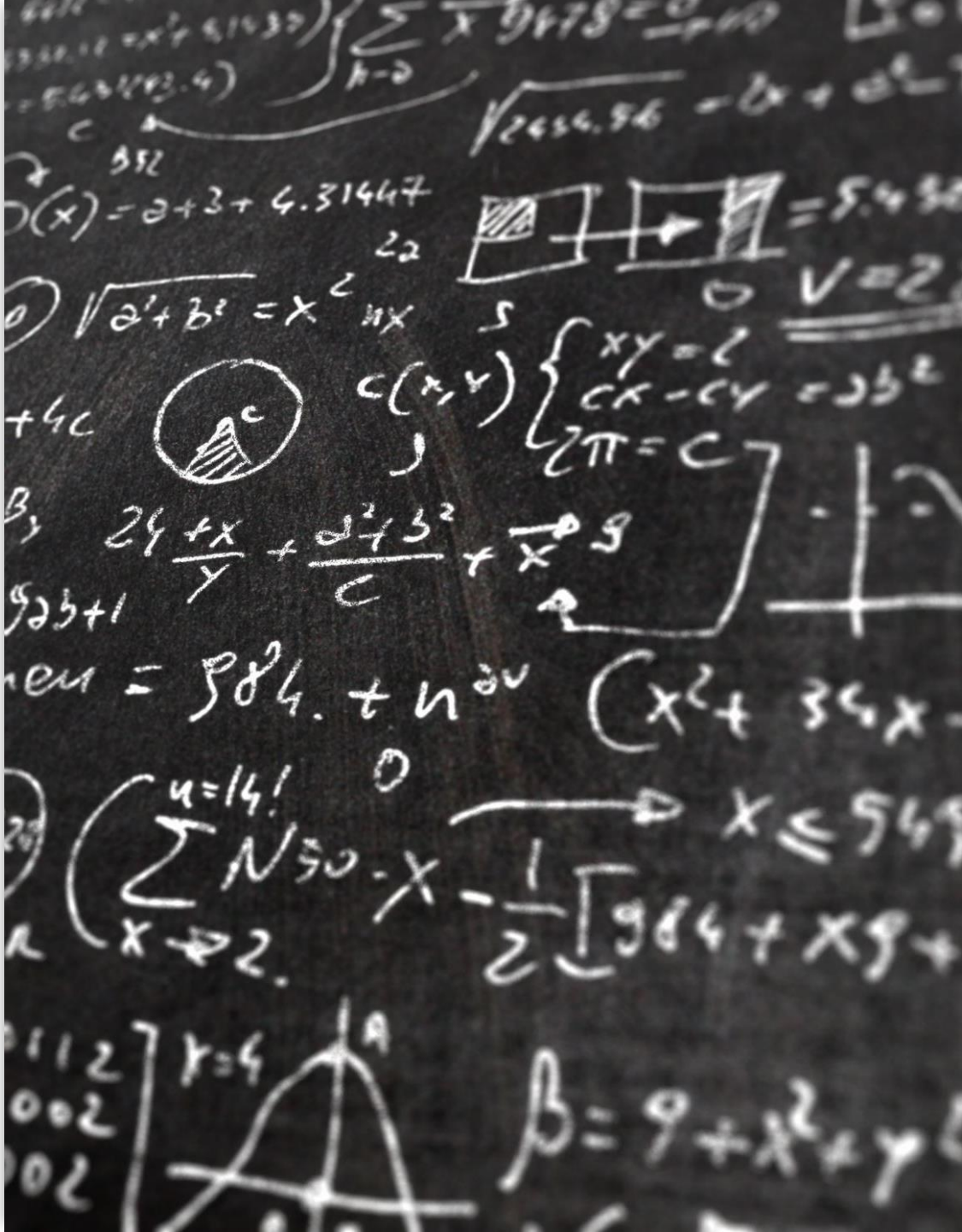
School of Computing & Communications | Lancaster University

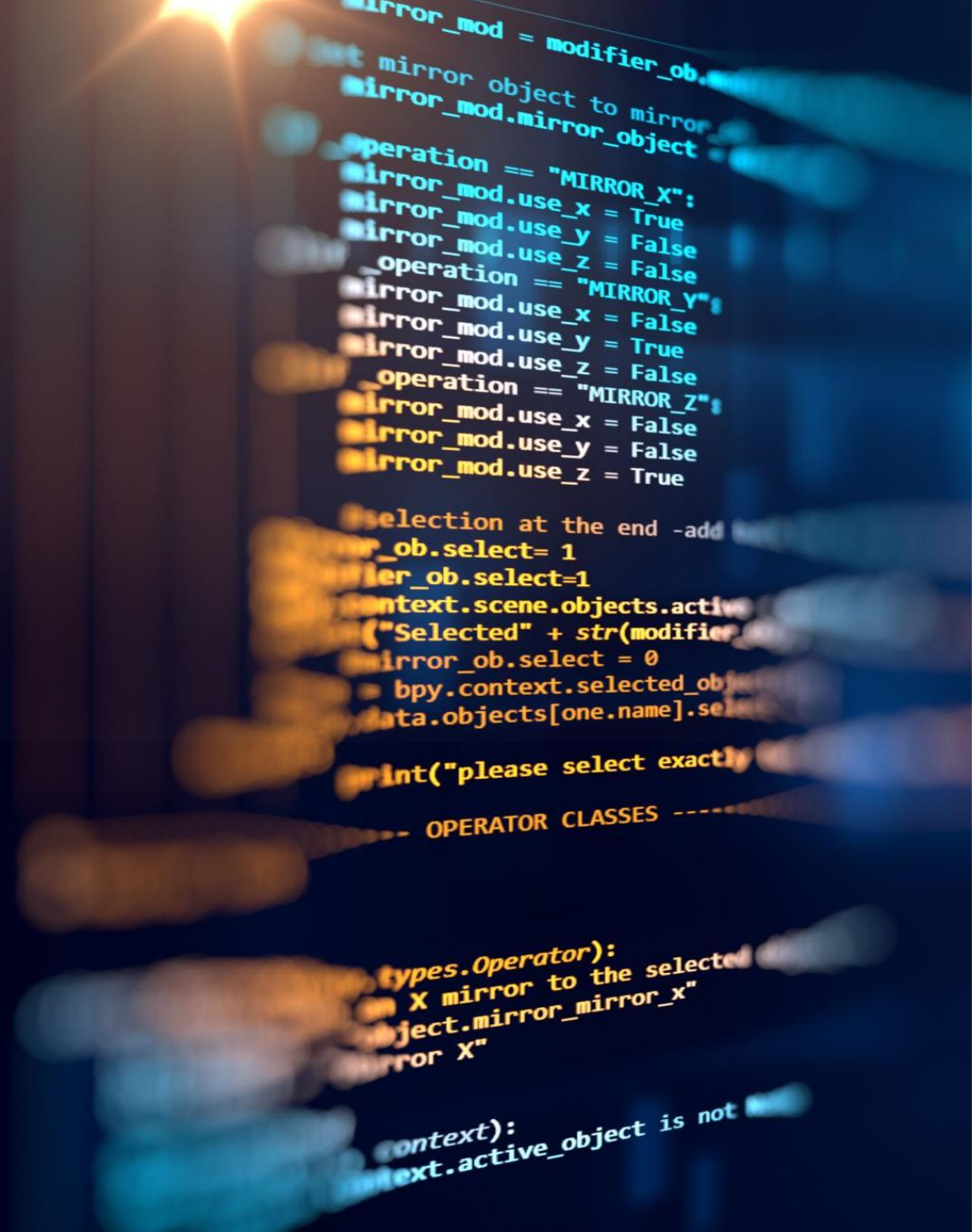
SCC.111 Software Development – Lecture 16: Multifile Projects

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

This lecture

- What happens when our code is too big for one file?
- What is a library in C? Can I create my own?





What's the problem?

- If we write 'monolithic' solutions, *all our code is one file*
- *Imagine 100M+ lines of Windows 11 source in just one file shared by the 1000s of dev team programmers!*

Pros:

- **Only one file** for the compiler to ‘parse’ to find all the functions
- The programmer also just has **one place** to look (if it’s in, it’s in, we can scroll, search etc.)
- Copying the project is passing around **one source file** (no dependencies)

Cons

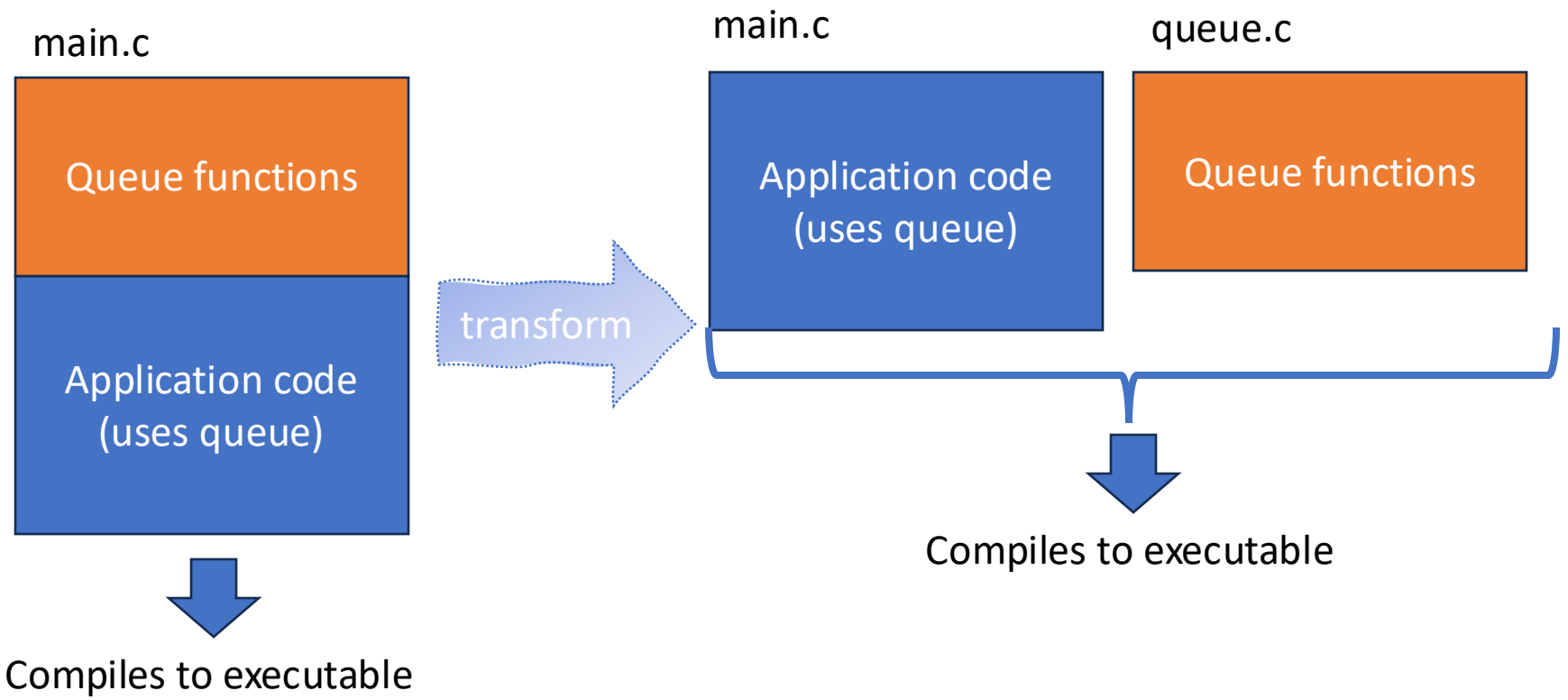
- Clearly **won't scale**, imagine a whole application in one file!
- Scrolling is eventually **unmanageable**
- Painful for **teamwork** and larger scale projects (co-editing the same file?)!
- Harder to package up functions (e.g. as an API or logical group) to **reuse** in other projects – would need to 'extract' the code
- Danger we have **hidden dependencies** and side effects (e.g. lack clean APIs and manage state directly, or worse, global variables/state)

What's the fix?

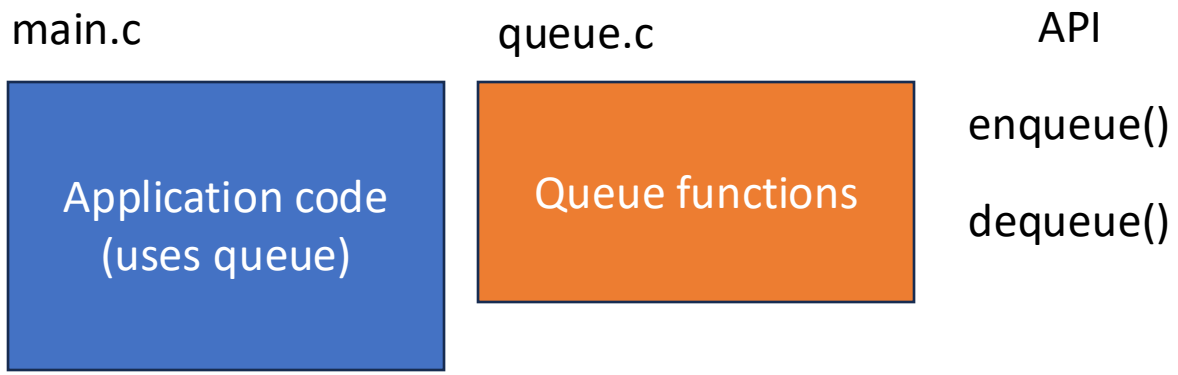
- Split projects into smaller units (i.e. multiple **.c** files)
- Create more but smaller source files
- Functions normally grouped by purpose
- Create useful sets of functions (cleaner APIs) that can be used and potentially reused across our projects

Abstract data types (ADTs) as an example

- Data structure handling (e.g. queue or set implementations) are a great example of reusable functionality that we can ‘split out’



Abstract data types (ADTs) as an example

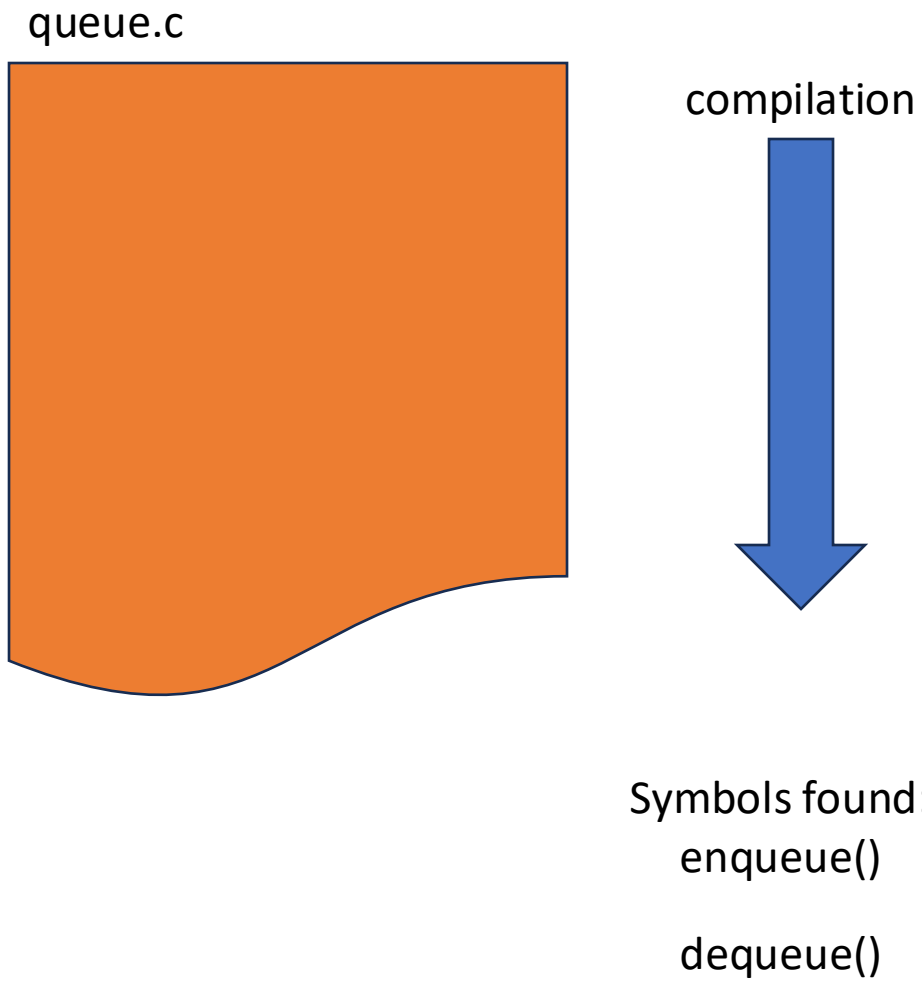


Functions ‘referenced’ and ‘called’ from main.c

Functions ‘declared’ and ‘exposed’ from queue.c

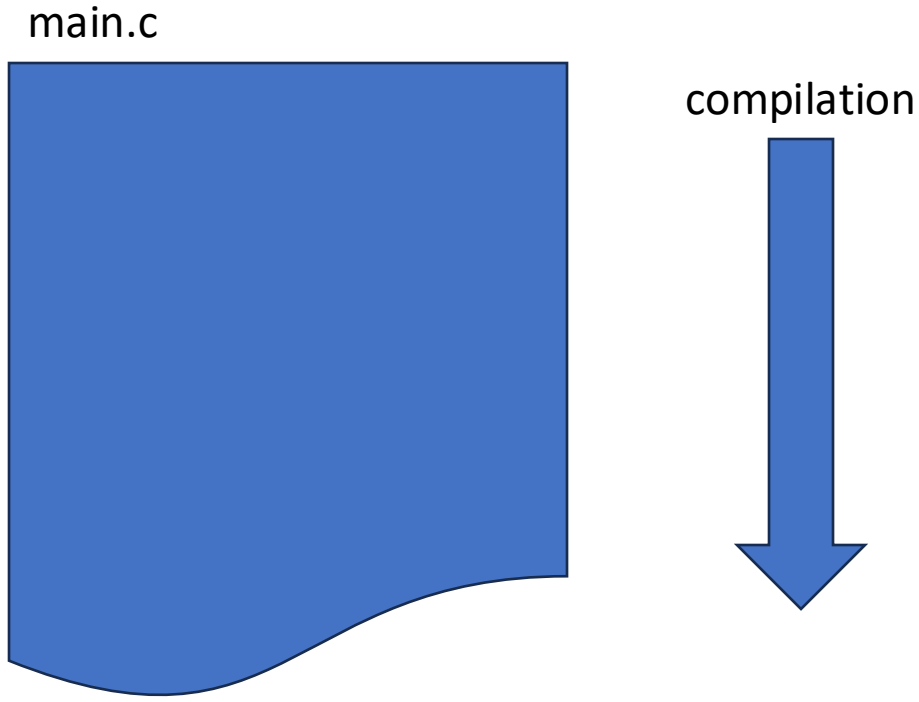
Consider the compiler of queue.c

- The compiler needs to know what functions exist
- How they can be called
- What their ‘signature’ is (name, arguments, return values)



Consider compiling main.c

- What are these functions not defined in main.c?
- What parameters do they take?
- Of what type?
- Ultimately, where is the code located so I can run it!



We need two things!

- To let main.c (or more generally, function callers) to know about the functions
- To tell the compiler that there are multiple parts to compile and assemble!

Forward declarations and header files (.h)

- As we've seen before, we can tell the compiler 'what to expect' for a function by declaring a forward declaration (prototype)

```
int move_forward(int howManyTimes);
```

- This is just the first line of a function terminated with a semi-colon (a function without a body)
- The forward declaration always needs to happen before the function is called

Example:

```
// Forward declaration here

int dequeue();

int main()
{
  // Calling dequeue from here

  dequeue();
}

// Implementation of dequeue here

int dequeue()
{
  …
}
```

Compiler learns about function spec here

Compiler knows what to expect here

Compiler actually declares function (that **must match** forward declaration) here

Example of splitting across files

main.c

```
// Forward declaration here

int dequeue();

int main()
{
  // Calling dequeue from here

  dequeue();
}
```

queue.c

```
// Implementation of dequeue here


int dequeue()
{
  ...
}
```

Less clumsily:

Includes

queue.h

```
// Forward declaration here

int dequeue();
```

main.c

```
#include “queue.h”

int main()
{
  // Calling dequeue from here

  dequeue();
}
```

queue.c

```
// Implementation of dequeue here

int dequeue()
{
  …
}
```

Note: #include “queue.h” not <>!

So now we have multiple C files

- We need to combine them into a single executable

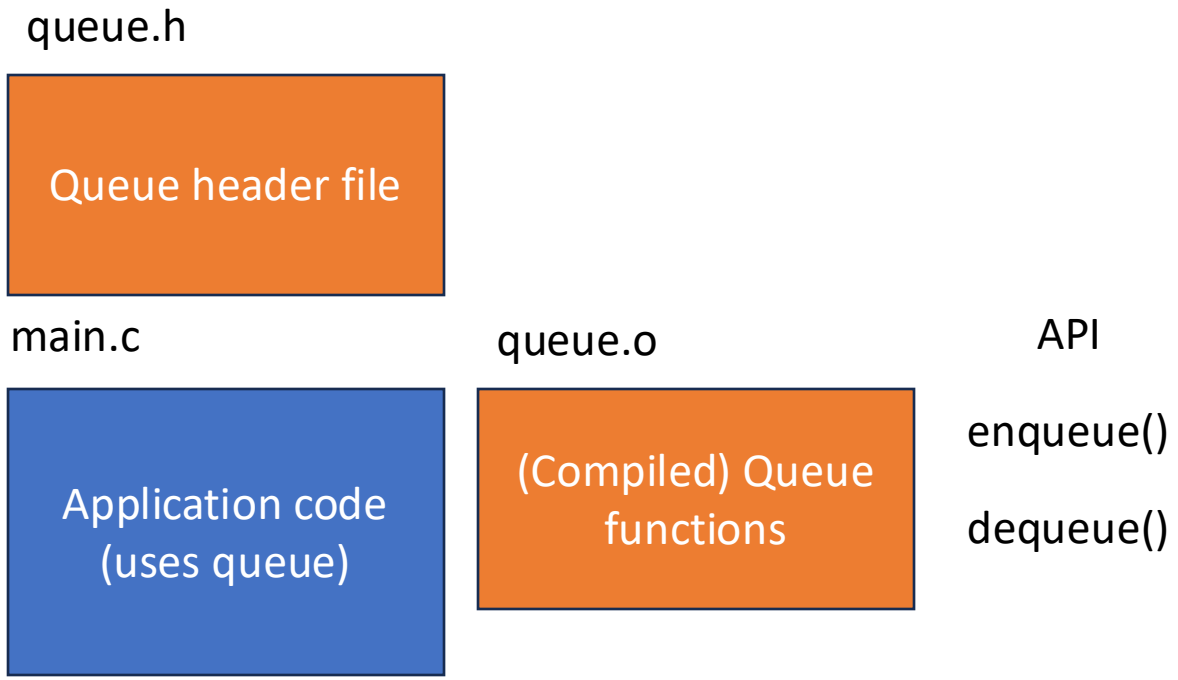
```
gcc -o target source1.c source2.c … <more source files>
```

e.g.

```
gcc -o main main.c queue.c
```

Can we pre-compile into a library?

- What if we want to share our code in compiled form (i.e. *not* give them the source)?



We just provide the **object** and **header** files

We can share the intermediate object file (.o)

- Creates ‘object/ part compiled file’ (binary)

```
gcc -c queue.c
```

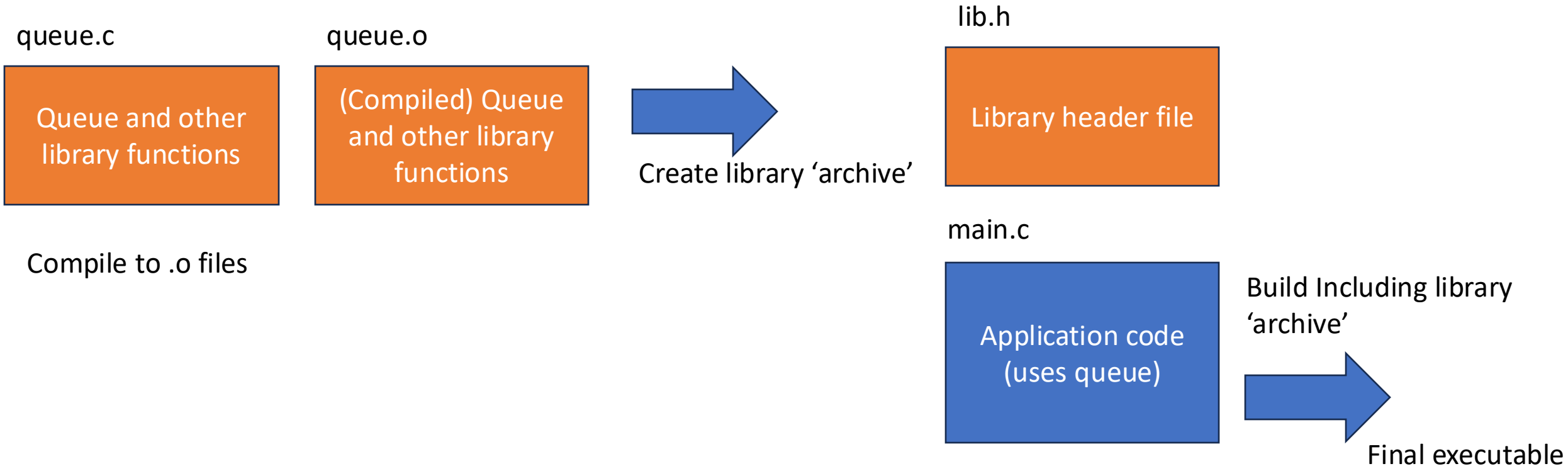
Creates queue.o

We need to combine them into a single executable

```
gcc -o main main.c queue.o
```

We might even package into a library

- One or more pre-compiled (object) files can be combined into a library!



Creating and using a static library

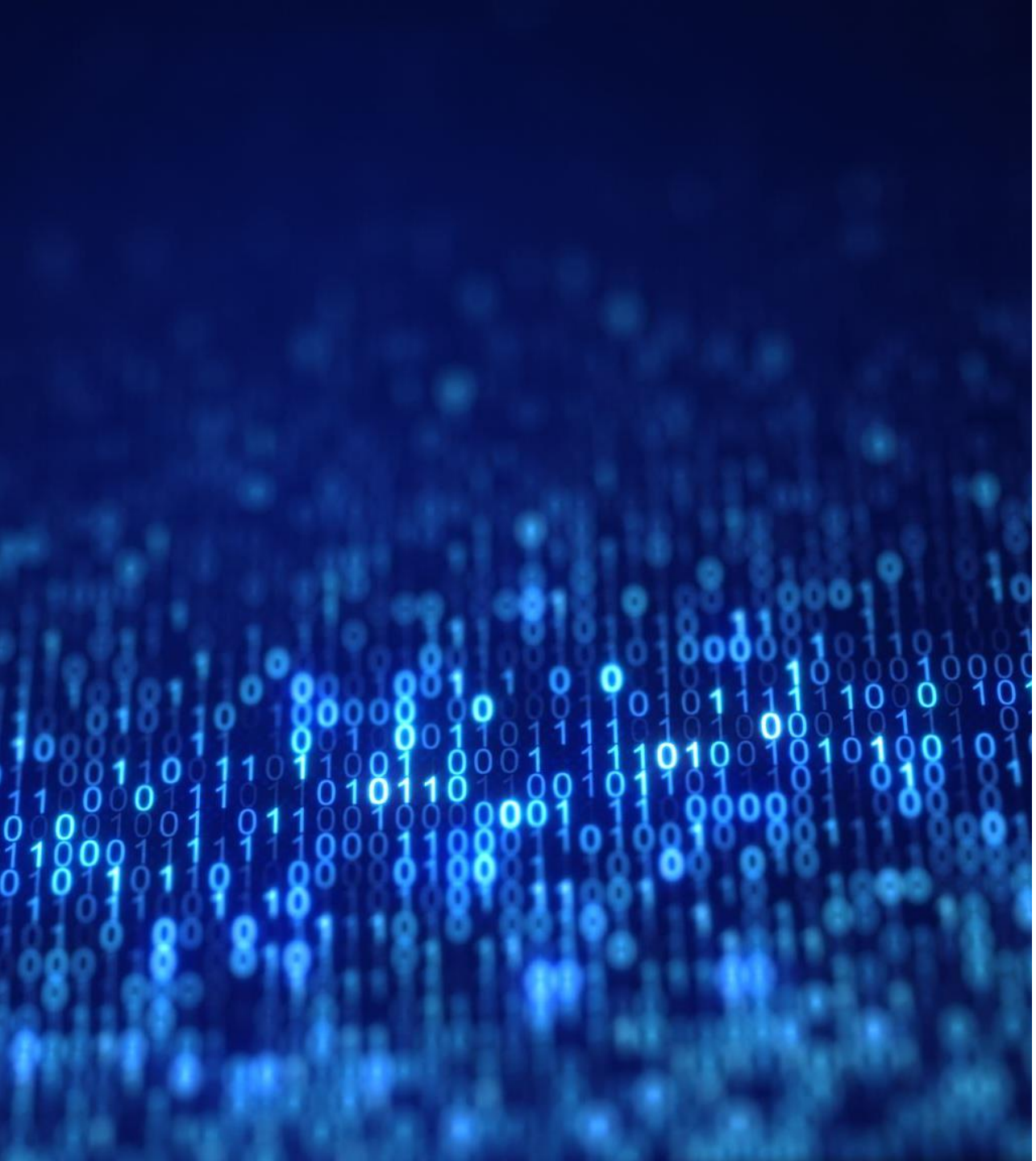
- We need to combine them into a single library (.a) or archive file

```
gcc -c queue.c
ar rcs libq.a queue.o
```

Combines compiled .o files (in .a archive) with source to create ‘main’

```
gcc -o main main.c –lq
./main
```

Note: -l option takes name of library without the leading ‘lib’ or trailing ‘.a’



Summary

- How forward declarations let us be flexible on where we declare functions
- Header files are sets of forward declarations for our library
- How we can compile multiple source files into one executable
- How we can pre-compile into libraries