Modules and Separate Compilation

Splitting Code Across Multiple Files

Writing programs whose source code resides in a single file doesn’t scale well

There’s nothing wrong with writing several hundred lines of C source code all in a single `.c` file, since the C compiler is sufficiently fast that recompiling such source files is almost instantaneous

However, should your project codebase grow to several thousand lines of source code, having it all in a single file causes compilation and linking to take noticeable time

Separate programmers in a team modifying different parts of the same file, may lead to difficulties merging

Some form of splitting the code into separately compiled C source files, with header files containing common declarations is a good idea.

How should we split code among many C source files?

When a program is split across multiple C source files and header files, the C compiler **compiles each C source file independently**, and then links all the object files together

According to modular conventions, most (but not all) of our code is structured as **modules**

In C, a module (called `wibble`, say) comprises a pair of C files: the source file `wibble.c`, and a corresponding header file `wibble.h`

Furthermore, `wibble.c` must always `#include "wibble.h"`, and every client of `wibble` must also `#include "wibble.h"`

Compilation: what requirements must we meet?

To successfully compile a single C source file (with no warnings), we must meet the following minimum compilation requirements:

- Every time you use a named type, that type must have been defined above, so that it’s size and memory layout is known. There is one exception to this - covered later
- Every time you use a function (calling it or taking it’s address), that function should have either been defined above (ie. a function with a body), or declared above (by a prototype)
- Every time you use a global variable (reading it, altering it, or taking it’s address), that variable should have either been defined above or declared above

When we say **declared or defined above** we mean that textually earlier in the source file, we either write the declaration/definition, or `#include` a header file containing that declaration/definition

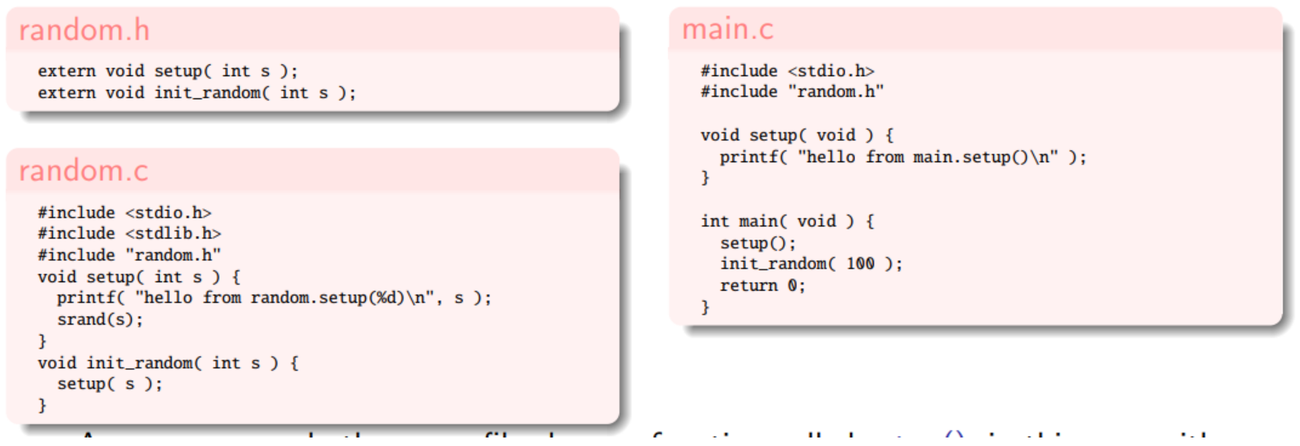
The C pre-processor makes those two options equivalent

Linking: what requirements must we meet?

To successfully link multiple object files together, we must meet the following minimum linking requirements:

- Every public symbol (function or global variable) must be defined precisely once. This includes `main()`
- If a symbol is declared but never defined then we get an undefined reference link error. Note that `main()` is implicitly declared
- If a symbol is defined more than once in separate object files, then we get a multiply defined symbol link error

Fixing a Function Name Clash



random.h

```
extern void setup( int s );
extern void init_random( int s );
```

random.c

```
#include <stdio.h>
#include <stdlib.h>
#include "random.h"
void setup( int s ) {
  printf( "hello from random.setup(%d)\n", s );
  srand(s);
}
void init_random( int s ) {
  setup( s );
}
```

main.c

```
#include <stdio.h>
#include "random.h"

void setup( void ) {
  printf( "hello from main.setup()\n" );
}

int main( void ) {
  setup();
  init_random( 100 );
  return 0;
}
```

Both source files have a function called `setup()`

You will get a compilation error (if the signatures mismatch) or a multiply defined symbol link error (if they are the same)

Private Helper Functions

In this case, the clashing setup functions are private helper functions

All private helper functions should be marked as static - meaning private to this C source file

As a special case, in a source file containing `main()`, you can usually mark every other function as static - because only `main()` can cause them to be called, they are all private

Behind the scenes, the C compiler gives each static function a new globally unique function name

In our example, add `static` to `random.c`'s `setup()` function and remove it's prototype from `random.h`

Can also make `main.c`'s `setup()` function `static`. So our code becomes:

random.h

```
extern void init_random( int s );
```

random.c

```
#include <stdio.h>
#include <stdlib.h>
#include "random.h"

static void setup( int s ) {
  printf( "hello from random.setup(%d)\n", s );
  srand(s);
}

void init_random( int s ) {
  setup( s );
}
```

main.c

```
#include <stdio.h>
#include "random.h"

static void setup( void ) {
  printf( "hello from main.setup()\n" );
}

int main( void ) {
  setup();
  init_random( 100 );
  return 0;
}
```

This now compiles and links successfully

Use `#include "filename.h"` for our own modules that are in the current directory, and `#include <filename.h>` for standard library header files

You can specify additional directories to search using the `-Idirectory` flag to `gcc`

- Several topics ago, we talked about scope and lifetime of variables - before we met separate compilation. In particular, we presented example scope/global.c:

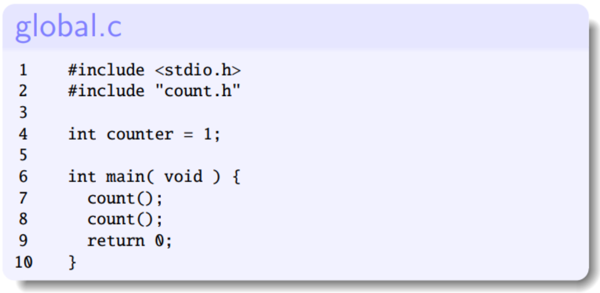
```
1   #include <stdio.h>
2
3   int counter = 1;
4
5   void count( void ) {
6     printf( "In count(): counter  = %d\n", counter );
7     counter++;
8   }
9
10  int main( void ) {
11    count();
12    count();
13    return 0;
14  }
```

Output

```
In count():  counter = 1
In count():  counter = 2
```

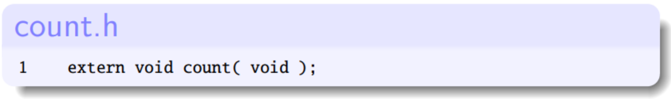
- We said that our global variable counter's scope reaches from the point it is defined (line 3) to the end of the source file (line 14).
- But what happens when we generalise this to separate compilation and linking of multiple source files?
- Can a global variable defined in one source file be used in another? If so, how do we extend our definition of scope to cope?

Consider this example, where we have taken our previous example `scope/global.c` and naively moved the `count()` function to another module:



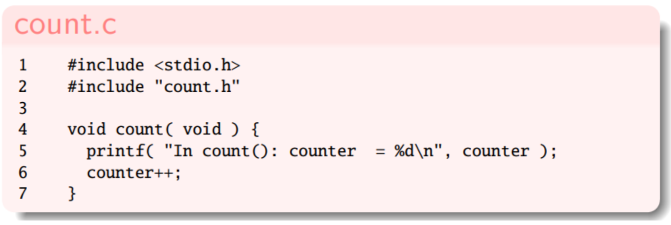
global.c

```c
1   #include <stdio.h>
2   #include "count.h"
3
4   int counter = 1;
5
6   int main( void ) {
7     count();
8     count();
9     return 0;
10  }
```



count.h

```c
1   extern void count( void );
```



count.c

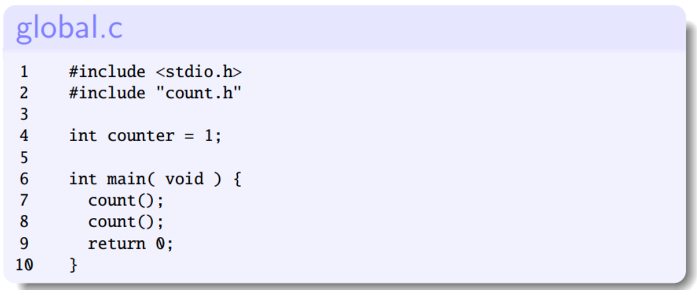
```c
1   #include <stdio.h>
2   #include "count.h"
3
4   void count( void ) {
5     printf( "In count(): counter  = %d\n", counter );
6     counter++;
7   }
```

Does this compile?

This does not compile

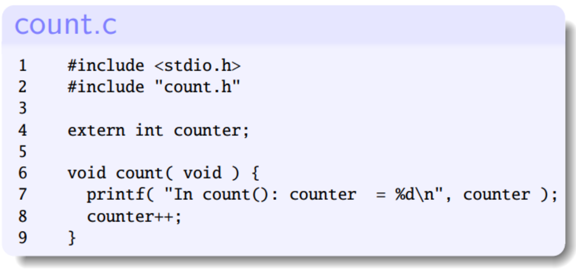
Each C source file is compiled separately, and has to make sense on it's own. In `count.c` there is no definition or declaration that tells the C compiler about `counter`

To declare that a global integer variable called `counter` exists somewhere, and to let us use it in `count.c`, add `extern int counter;`



global.c

```c
1   #include <stdio.h>
2   #include "count.h"
3
4   int counter = 1;
5
6   int main( void ) {
7     count();
8     count();
9     return 0;
10  }
```



count.c

```c
1   #include <stdio.h>
2   #include "count.h"
3
4   extern int counter;
5
6   void count( void ) {
7     printf( "In count(): counter  = %d\n", counter );
8     counter++;
9   }
```

This compiles!

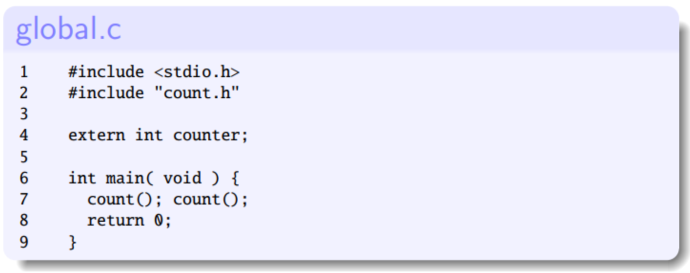
What is the scope of `counter` now? `counter` is visible in `global.c` lines 4-10 and in `count.c` lines 4-9

The lifetime of `counter` is unchanged

The `extern` keyword is compulsory here

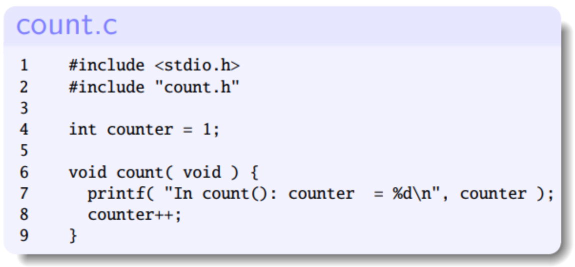
Question: Who should 'own' `counter`?

We should have moved `count()` and `counter` together, meaning that `count.c` would own `counter`. Let's try that:



global.c

```c
1    #include <stdio.h>
2    #include "count.h"
3
4    extern int counter;
5
6    int main( void ) {
7      count(); count();
8      return 0;
9    }
```



count.c

```c
1    #include <stdio.h>
2    #include "count.h"
3
4    int counter = 1;
5
6    void count( void ) {
7      printf( "In count(): counter  = %d\n", counter );
8      counter++;
9    }
```

In this case, `global.c` doesn't need to access `counter` at all, so we can remove `extern int counter;` from `global.c`, reducing `counter`'s scope to `count.c` lines 4-9

Sharing Global Variables - a Critique

You should have as few shared globals as possible

However, on occasion a shared global variable may be the most elegant and efficient thing to do, especially when there is a singleton data structure variable which underpins your entire system

In this case, you don't want to pass that data structure variable around in every function until you need it

But the OO solution may not appeal either: wrapping that data structure variable away inside a module and providing access functions to extract parts of it, and update functions to make important changes to it (in a controlled way)

The most efficient solution may well be to share the global variable into whichever C source files you need it in, perhaps using those update functions to modify it

Static Global Variables - private globals

In our previous example, our global variable `counter` was only needed in `count.c`

This can be enforced by making `counter` a static global variable: a global variable private to the current C source file, created by defining it `static` and global (outside all functions):

```c
#include <stdio.h>
#include "count.h"

static int counter = 1;

void count( void ) {
  printf( "In count(): counter  = %d\n", counter );
  counter++;
}
```

Scope: from the point of definition to the end of the current source file

Lifetime: Global

Just like static functions, behind the scenes the C compiler gives it a unique name

How should we split code among many C source files?

One bad practice is using `#include` to include one .c file into another. **You should only ever include `.h` files into `.c` files**

Furthermore, `.h` files should never contain function or global variable definitions, only function and global variable declarations, plus constants and types

Splitting C source files: a worked example

The first rule of software engineering is **Don’t Repeat Yourself**

If you have a block of **identical code** in multiple places, extract it out into a module, and use that module in all necessary places

If you have a block of **almost identical code** in multiple places, extract it out into a module, and put the code that is different in another function
In the module c file, create a declaration using `extern`. In each of the files that use this module, define the function according to their specific implementations

Make all variables and functions that are not part of the module's service `static`

`sym.[ch]`: Our first C module

`sym.[ch]` is our first proper C module. It shows us the C modular conventions:

- The .h file describes the public interface of the module - it contains `#defines` and type definitions for any public types and prototypes of all public functions
- The corresponding `.c` file implements that interface - it provides a definition for each public function which had a prototype in the `.h` file - plus any number of static functions (private helpers), along with any number of static global variables
- We `#include "sym.h"` at the top of `sym.c`, as well as in any other C source file that uses any of `sym`'s types or public functions
    - We do this so that the C compiler will warn us if any of our function definitions don't match the prototypes in the header file

Tips and Techniques picked up from Example Modules

Keep variables private to the files they are declared in, and write getter and setter functions.

The `.h` files are all very short, this is typical in C

`main()` is typically very short and simple. If its not, break the code out into a series of helper functions

You could split large files into **high level functions** and **low level helper functions**; put the low level helper functions into a separate file

This splits up large files, however, the helper functions will now no longer be static

There's no single "correct" modular decomposition of your code. There's more than one way to do it!

Place a comment immediately above each prototype in each header file

You can copy and paste these comments immediately above each function in the corresponding `.c` file

Nested Include

Placing a `#include` inside a header file

Such code is very fragile

First, we now have the possibility of creating an include loop

Second, we might `#include` the same header file twice in a file by accident.
Header files are designed to be included once per C source file

Before we allowed ourselves to nest includes, that was always true - now it may not be

How do we solve this?

Include Guards

A conventional idiomatic solution has emerged to solve this: the content of a **header file** can be surrounded by **include guards**. They have the form:

```
#ifndef __SOME_UNIQUE_SYMBOL__
#define __SOME_UNIQUE_SYMBOL__
// The original content of the header file goes here
#endif
```

The symbol must be unique across your codebase

It represents the idea that the content of this header file has already been included

The usual convention is `uppercase(module name)+ “_H”`

So for module `coord` we have `COORD_H`
An Example:

```
#ifndef COORD_H
#define COORD_H

typedef struct { double x,y; } coordinate;

extern coordinate make_coordinate( double, double );
```

```
#endif
```

Do this for all Header files in your codebase

Now, it works!

`#ifndef...#endif` is a rarely used feature of the C pre-processor that allows conditional compilation. `#ifndef` means if the symbol is not defined

Simulating Namespaces

C doesn’t support namespaces, or polymorphism

Therefore, having identically named functions, or types, or global variables in two different C source files in the same codebase causes a compilation or linking error

We saw that making private helper functions, and private global variables, `static` fixed many clashes

But what if both functions/globals are public?

The systematic way to fix this problem is to qualify the names of public functions, types, and global variables - add the module name to them

You should choose whether you’re going to always place the module name first or last

Modules and Separate Compilation - Summary

- Ok, we've covered quite a lot of ground there. To recap:
- You should always use the modular conventions in C:
  - Structure most of your code as modules,
  - A module called wibble comprises a pair of files: a header file `wibble.h` - the interface, and a source file `wibble.c` - the implementation.
  - `wibble.c` must always `#include "wibble.h"` to check consistency.
  - All users or clients of module wibble must also `#include "wibble.h"`.
- It's perfectly fine to have isolated ".h" files, I often have project-wide common definitions in defns.h. Ditto for isolated ".c" files - eg. every main program.
- One particularly common class of modules is called an Abstract Data Type, and gathers a type and all the operations on that type together. Often the type has the same name as the ADT module.
- Mark all public functions with the module name. Make as many helper functions static (private) as possible. Make global variables static wherever possible. Define everything exactly once.