CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Compilation steps
- Review pre-processor
- Header files
  - Includes
  - Definitions

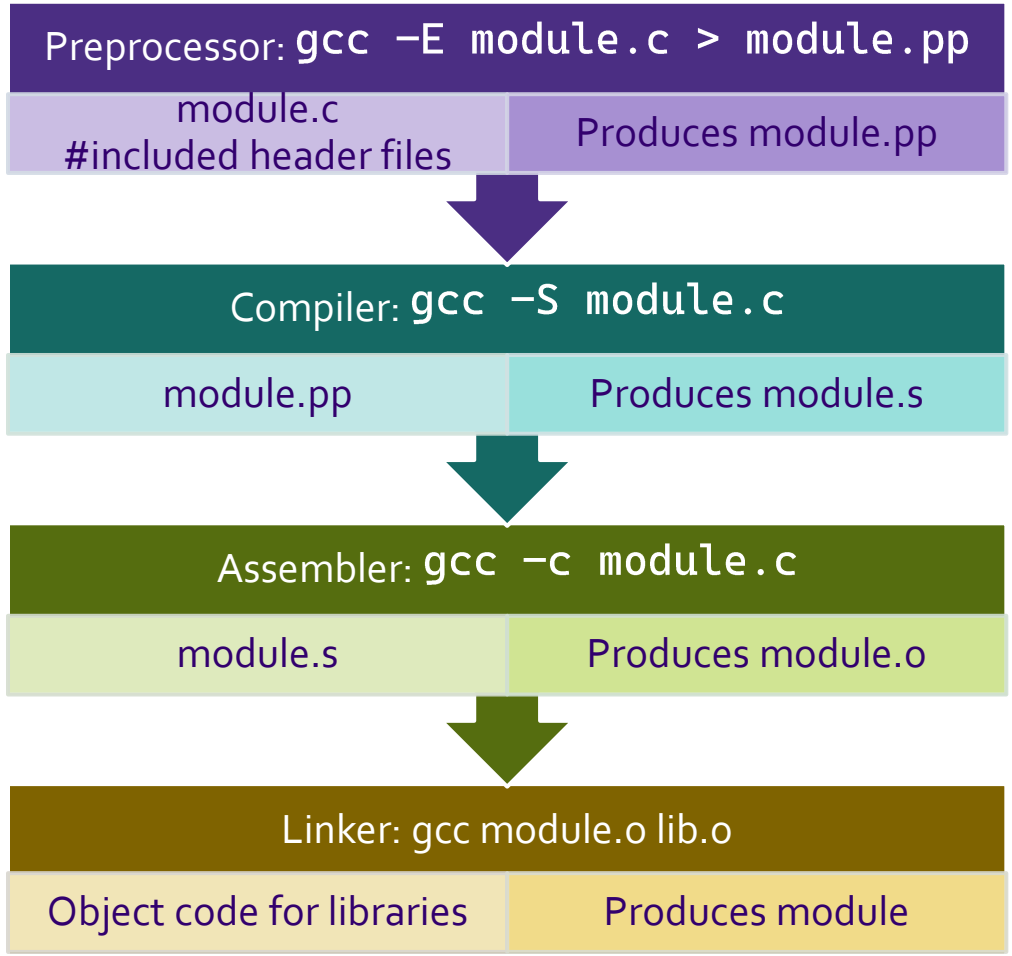
Your Goals

- Debug interview scheduling
- HWs 4&5
- Check Canvas

Compilation

- Compiling a module is a multi-step process
- What happens when we combine multiple modules?
- Let's explore!

Compiling process



- Each step feeds into the next
- You can stop the process at any time
  - And do: when building a large project you often build object code and link it in separate steps.

Preprocessor

- Preprocessor rewrites the code before the compiler gets it
  - Includes header files
  - Defines constants
  - Defines macros
  - Conditional compilation

Preprocessor - includes

- Preprocessor rewrites the code before the compiler gets it
  - Includes header files
  - Defines constants
  - Defines macros
  - Conditional compilation

```
#include <stdlib.h>
#include “userfile.h”
```

Header files (use ‘.h’),

Headers include function, struct, constant declarations

Never include function implementations

Never include ‘.c’

`$gcc -l` : look in specific directories

#includes copies included files to code

What to include in a header file

- **You create a `.h` file to share code with another calling module**
  - Declare any variables and functions you want another caller to user
  - Functions you want to use only in the same file are declare in the .c file
  - Include libraries needed to compile the header file
  - Libraries only needed to compile the .c module are included in the .c file
- If you have `a.c`, which uses `printf include <stdio.h>` in `a.c`
- If you also have `b.c`, which uses `printf`, you could include `<stdio.h>` in `a.h` and not in `a.c` or `b.c`, however
- Generally, include any header files needed for directly-called functions (promotes encapsulation), so `b.c` would include `<stdio.h>`

Encapsulation

- One of the four pillars of OOP
- Bundling of data and the mechanisms that operate on that data
  - Promotes cohesion – keeps related data and mechanisms together
  - 'hides' details of storage or mechanisms for security or efficiency
  - Decouples interface from implementation
- Pragmatically programmers isolate one component of the system
  - Easier to design and test (work to a limited API)
  - Easier to maintain (update without having to fix external code)

Declare

Specifies Name & Datatype

No memory allocated

Can happen many times

```
extern int x;
int fun(float a);
```

Define

Allocates Memory

Functions get body

Can only happen once

```
int x;
int fun(float a)
{return (int)a;}
```

Initialize

Gives a value

Must be defined (have memory)

Could be re-set

```
int x = 10;
```

`#ifndef` - header file inclusion

```
#ifndef FOO_H
#define FOO_H
```

and end it with:

```
#endif
```

- Assuming nobody else defines SOME_HEADER_H (convention)
  - first #include "some_header.h" will do the define and include the rest of the file
  - second and later will skip everything
- More efficient than copying the prototypes over and over again
- In presence of circular includes, necessary to avoid "creating" an infinitely large result of preprocessing

Symbolic Constants

- Creates TOKEN to represent more text
- Preprocessor:
  - Replaces all matching TOKENS in rest of file
  - Knows where words start and end
  - Has no notion of scope (not the compiler)
- Can shadow another `#define`
- Use `#undef` to remove

```
#define SYMBOLIC_CONSTANT value
#define NOT_PI 22/7
#define VERSION 3.14
#define FEET_PER_MILE 5280
#define MAX_LINE_SIZE 5000
```

Macros

- Still string replacement
- Gotchyas with argument expansion
- Not needed for performance overhead (anymore)
- Can be flexible (no data type required)

```
#define TWICE_AWFUL(x) x*2
#define TWICE_BAD(x) ((x)+(x))
#define TWICE_OK(x) ((x)*2)

double twice(double x) {  return x+x;  }


y=3;
z=4;
w=TWICE_AWFUL(y+z);    [y+z*2]
z=TWICE_BAD(++y);      [++y + ++y]
z=TWICE_BAD(y++);      [y++ + y++]
```

Macros - Debugging

- Remember – pure string replacement
- Demo twice.c

```
#define TWICE_AWFUL(x) x*2
#define TWICE_BAD(x) ((x)+(x))
#define TWICE_OK(x) ((x)*2)

double twice(double x) {  return x+x;  }


y=3;
z=4;
w=TWICE_AWFUL(y+z);    [y+z*2]
z=TWICE_BAD(++y);      [++y + ++y]
z=TWICE_BAD(y++);      [y++ + y++]
```

Good Macros

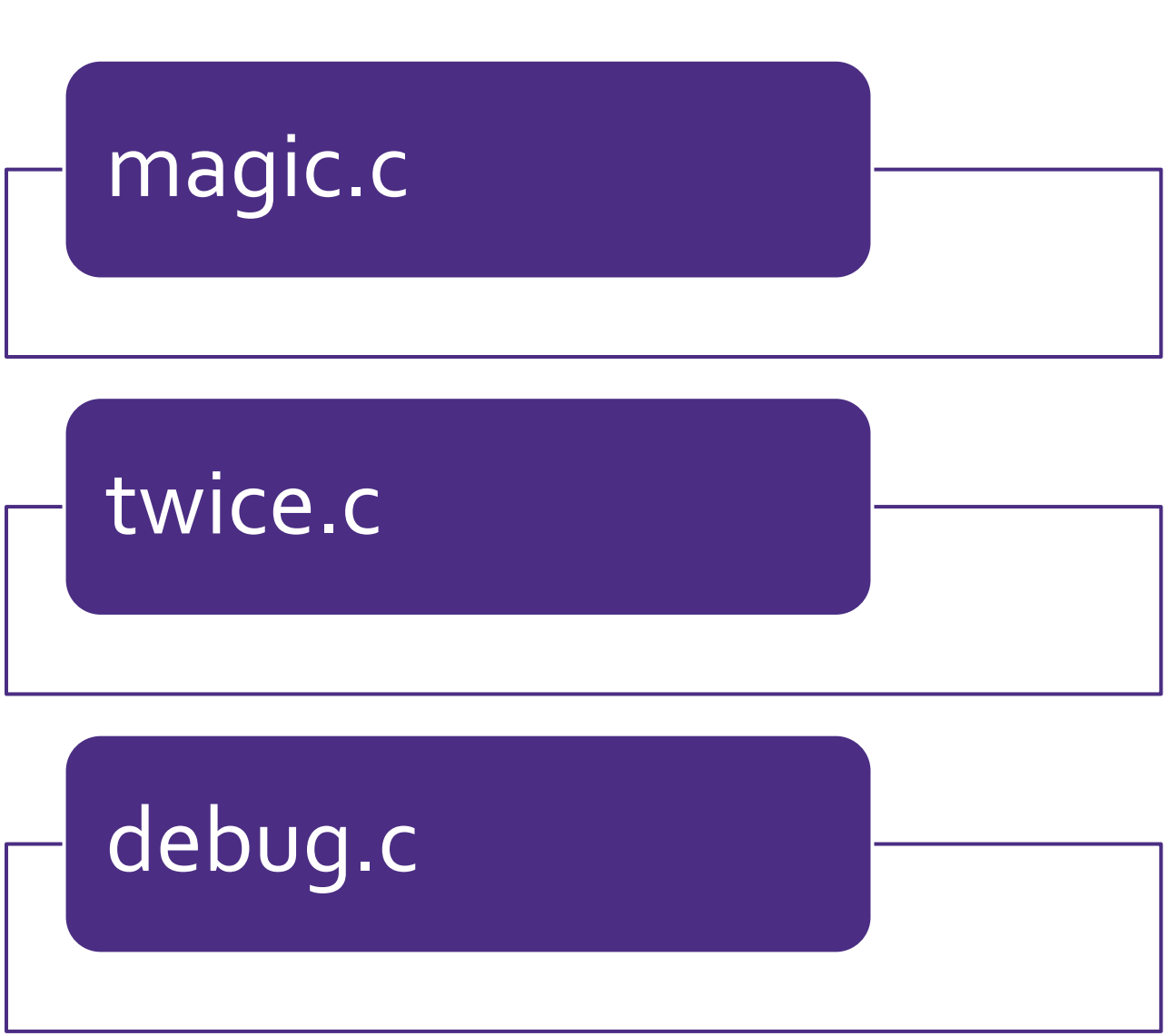
- Parameterized macros are generally to be avoided (use functions)
- **BUT,** there are things functions cannot do:
- **#define NEW_T(t, howmany) ((t*)malloc((howmany)*sizeof(t))**
- **#define PRINT(x) printf("%s:%d %s\n", __FILE__, __LINE__,x)**

  Be very careful with syntax if you do use them

Conditional Compilation

```
#ifdef FOO
// only compiled if FOO is defined
#endif

#ifndef FOO
// only compiled if NOT FOO
#endif

#if FOO > 2
// only compiled if FOO > 2
#endif
```

```
// use DBG_PRINT for debug-printing
#ifdef DEBUG
#define DBG_PRINT(x) printf("%s",x)
#else
// replace with nothing
#define DBG_PRINT(x)
#endif

DBG_PRINT("hello world!\n");
$ gcc -D DEBUG foo.c
// or with #define
```



#define

Global variables

- Declared with normal syntax, but outside any functions
- Must be declared within file to be 'known' (could be put in header).

```
#include <stdio.h>

#define TWICE_AWFUL(x) x*2
#define TWICE_BAD(x) ((x)+(x))
#define TWICE_OK(x) ((x)*2)

int ex_global;

int main(int argc, char **argv) {
```

Static variables & extern

- Global variables have space allocated in the global memory section, not the stack.
  - Persist and can be used by all the functions within scope: the same source file
  - UNLESS, keyword extern is used
  - To use a global variable across multiple modules put extern declaration in the header file

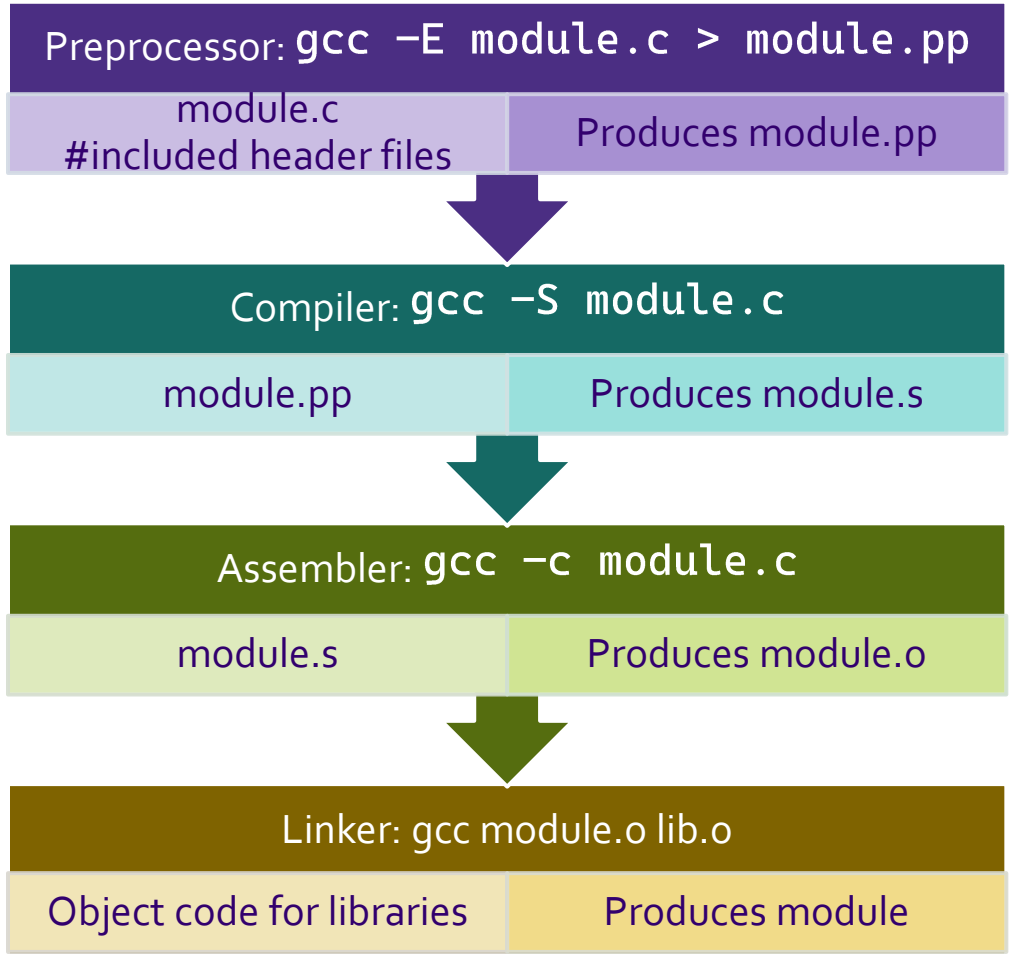
```
extern int var;   // in header
int var = 0;      // in file

int main(void) {
  var = 10;
  return 0;
}
```

- C keyword static allocates space in the global memory section, not the stack.
  - Memory persists outside of scope
  - Can not have a static variable in a struct

```
int fun() {
  static int count = 0;
  count++;
  return count;
}
```

- A static function limits the scope of the function
  - Only called within the same source file
  - Allows for encapsulation

Compiling process review



- Each step feeds into the next
- You can stop the process at any time
  - And do: when building a large project you often build object code and link it in separate steps.