
CSE 374

Programming Concepts & Tools

Hal Perkins

Winter 2017

Lecture 13 – C: The Rest of the Preprocessor

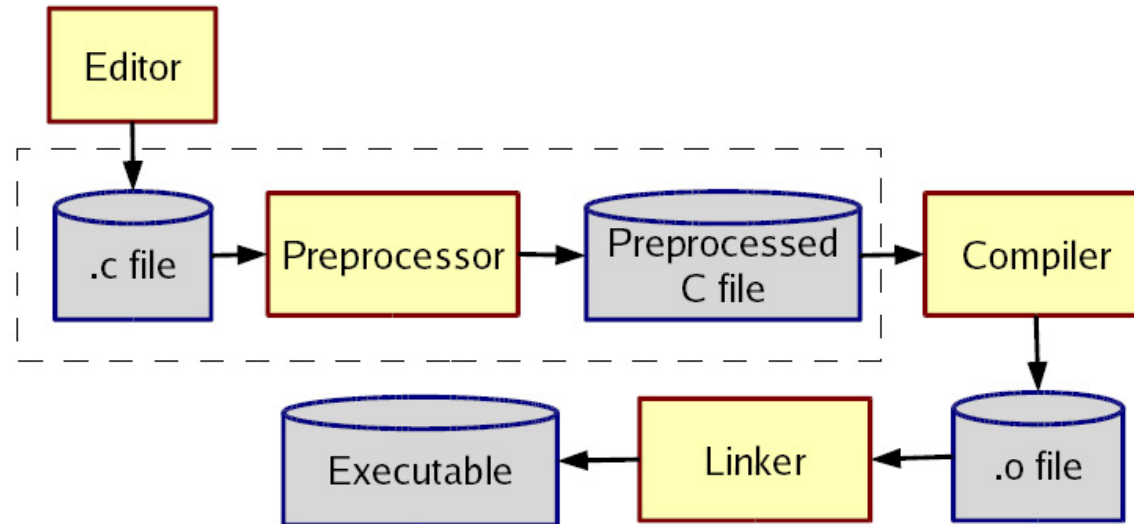
Administrivia

- Midterm should be graded this weekend and returned next Monday (or, at least, that's the plan)
- HW5 due ~~Tuesday~~ Thursday night
 - Questions? Need to look at demo again?
- HW6 – multiple parts with a partner
 - Pick a partner by early next week. Partner info *must* be submitted online by 11pm Wednesday (details on how/where in a couple of days)

The story so far...

- We've looked at the basics of the preprocessor
 - #include to access declarations in header files
 - #define for symbolic constants
- Now:
 - More details; where it fits
 - Multiple source and header files
 - A bit about macros (somewhat useful, somewhat a warning)

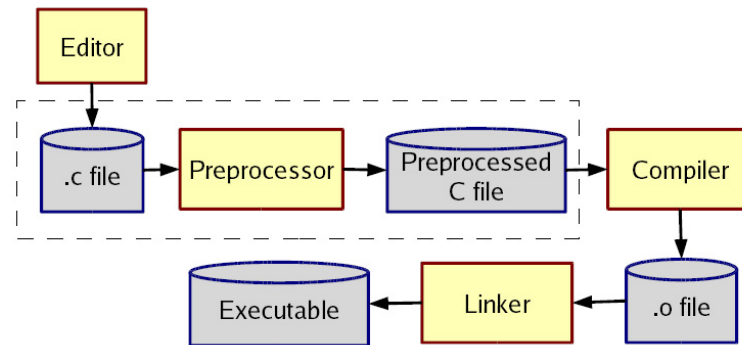
The compilation picture



gcc does all this for you (reminder)

- -E to only preprocess; result on stdout (rare)
- -c to stop with .o (common for individual files in larger program)

More about multiple files



Typical usage:

- Preprocessor `#include` to read file containing declarations describing code
- Linker combines your `.o` files *and* other code
 - By default, the “standard C library”
 - Other `.o` and `.a` files
 - Whole lecture on linking and libraries later...

The preprocessor

- Rewrites your .c file before the compiler gets at the code.
 - Lines starting with # tell it what to do
- Can do crazy things (please don't); uncrazy things are:
 1. Including contents of header files
 2. Defining constants and parameterized macros
 - Token-based, but basically textual replacement
 - Easy to misdefine and misuse
 3. Conditional compilation
 - Include/exclude part of a file
 - Example uses: code for debugging, code for particular computers (handling portability issues), “the trick” for including header files only once

File inclusion (review)

`#include <hdr.h>`

- Search for file `hdr.h` in “standard include directories” and include its contents in this place
 - Typically lots of nested includes, result not fit for human consumption
 - Idea is simple: declaration of standard library routines are in headers; allows correct use after declaration

`#include "hdr.h"`

- Same, but first look in current directory
- How to break your program into smaller files that can call routines in other files
- `gcc -I` option: look first in specified directories for headers (keep paths out of your code files) (not needed for 374)

Header file conventions

Conventions: always follow these when writing a header file

1. Give included files names ending in .h; only include these header files. **Never** #include a .c source file
2. Do not put functions definitions in a header file; only struct definitions, prototypes (declarations), and other includes
3. Put all your #includes at the beginning of a file
4. For header file foo.h start it with:

```
#ifndef FOO_H
```

```
#define FOO_H
```

and end it with:

```
#endif
```

(We will learn why very soon)

Simple macros (review)

Symbolic constants and other text

```
#define NOT_PI 22/7
```

```
#define VERSION 3.14
```

```
#define FEET_PER_MILE 5280
```

```
#define MAX_LINE_SIZE 5000
```

- Replaces all matching *tokens* in rest of file
 - Knows where “words” start and end (unlike sed)
 - Has no notion of scope (unlike C compiler)
 - (Rare: can shadow with another #define or use #undef to remove)

Macros with parameters

```
#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; } // best (editorial opinion)
```

- Replace all matching “calls” with “body” but with text of arguments where the parameters are (*just* string substitution)
- Gotchas (understand why!):
 - `y=3; z=4; w=TWICE_AWFUL(y+z);`
 - `y=7; z=TWICE_BAD(++y); z=TWICE_BAD(y++);`
- Common misperception: Macros avoid performance overhead of a function call (maybe true in 1975, not now)
- Macros can be more flexible though (TWICE_OK works on ints and doubles without conversions (which could round))

Justifiable uses

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)
```

Conditional compilation

`#ifdef FOO` (matching `#endif` later in file)

`#ifndef FOO` (matching `#endif` later in file)

`#if FOO > 2` (matching `#endif` later in file)

(You can also have a `#else` inbetween somewhere.)

Simple use: `#ifdef DEBUG // do following only when debugging`
`printf(...);`
`#endif`

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

- Note: `gcc -D FOO` makes FOO “defined”

Back to header files

- Now we know what this means:

```
#ifndef SOME_HEADER_H
#define SOME_HEADER_H
... rest of some_header.h ...
#endif
```
- Assuming nobody else defines `SOME_HEADER_H` (convention), the first `#include "some_header.h"` will do the define and include the rest of the file, but the 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
- So we always do this

C preprocessor summary

- A few easy to abuse features and a bunch of conventions (for overcoming C's limitations).
 - #include (the way you say what other definitions you need; cycles are fine with “the trick”)
 - #define (avoids magic constants; parameterized macros have a few justifiable uses; token-based text replacement)
 - #if... (for showing the compiler less code)