

# CSE 303: Concepts and Tools for Software Development

Hal Perkins

Autumn 2007

Lecture 12— C: The C Preprocessor; printf/scanf

## Where are We

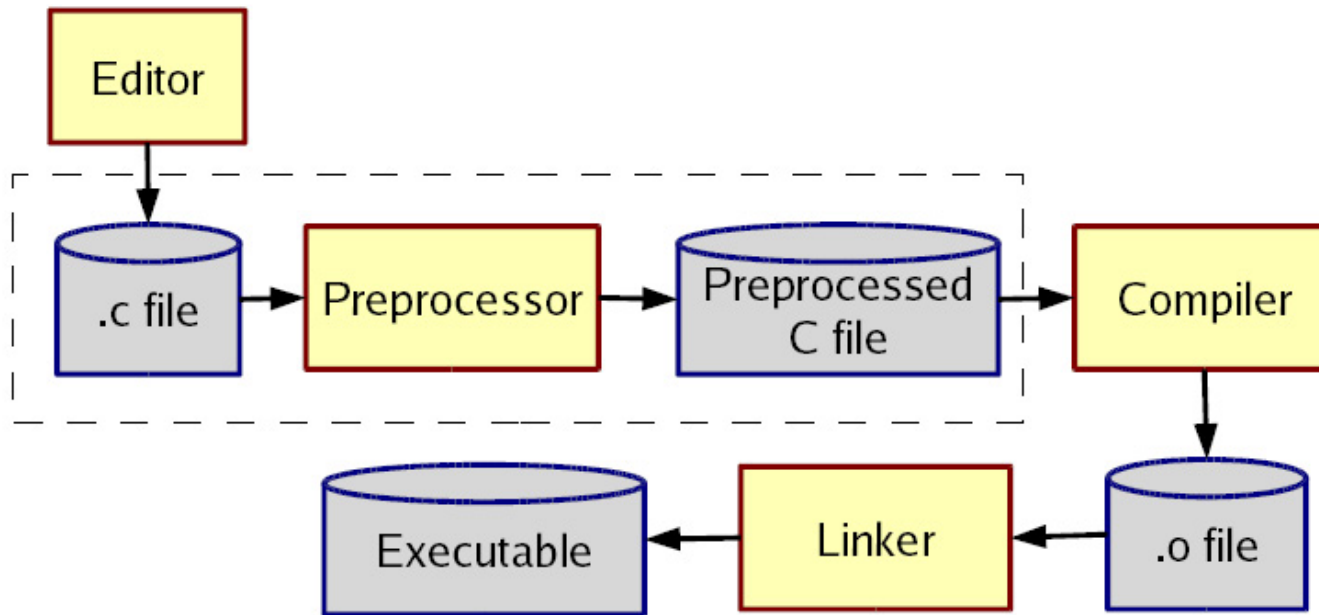
---

Two important “sublanguages” used a lot in C (almost every program)

- The preprocessor: runs even before the compiler (hence the name)
- printf/scanf: interpret certain strings funny at run-time
  - Really just a library though

Two lectures in one (preprocessor a bigger topic).

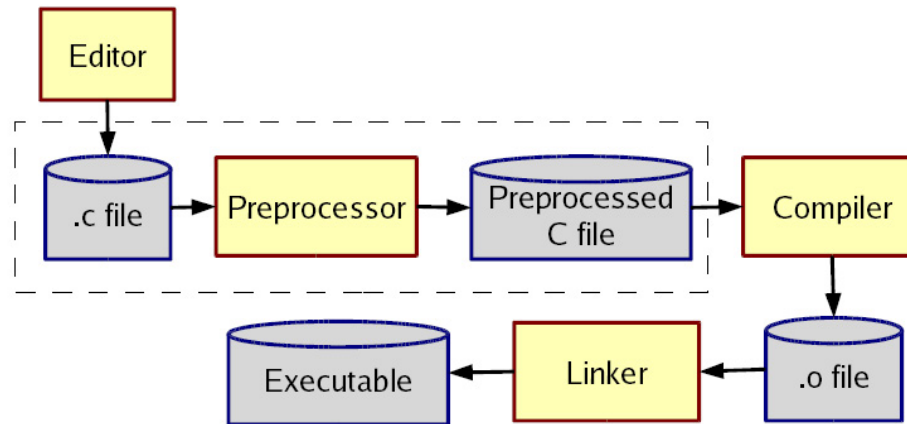
# The compilation picture



gcc does all this for you

- -E to only preprocess, put result on stdout (rare)
- -c to stop with .o (common; for part of a program)

# More about multiple files



Typical usage:

- Preprocessor `#include` to get a file describing *prototypes* (just types of functions/variables, *not* code)
- Linker is passed your `.o` and other *code*
  - By default, the “standard C library”
  - Other `.o` and `.a` files (hence `-lm` in homework 3)

Whole lecture on the linker 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 (see previous slide)
2. Defining *constants* and *parameterized macros* (textual-replacements)
  - Actually token-based (to be explained)
  - Easy to misdefine and misuse
3. Conditional compilation
  - Include/exclude part of a file
  - Example uses: code for debugging, code for some computers, “the trick” for including header files only once

# File inclusion

---

```
#include <foo.h>
```

- Search for file `foo.h` in “system include directories” (on `attu` `/usr/include` and subdirs) for `foo.h` and include its *preprocessed* contents (recursion!) at this place.
  - Typically lots of nested includes, so result is a mess nobody looks at.
  - Idea is simple: declaration for `fgets` is in `stdio.h` (use `man` for what file to include)
- `#include "foo.h"` the same but *first* look in current directory.
  - How you break your program into smaller files and still make calls to other files.
- `gcc -I dir1 -I dir2 ...` look in these directories for all header files first (keeps paths out of your code files).

# Conventions

---

*Conventions* to always follow:

1. Give included files names ending in `.h`; only include these *header* files.
2. Do *not* put functions in a header file; only struct definitions, prototypes, and other includes
3. Do all your `#include` 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 soon.)

## Simple macros

---

```
#define M_PI 3.14 // capitals a convention to avoid problems
```

```
#define DEBUG_LEVEL 1
```

```
#define NULL 0 // already in standard library
```

Replace all matching *tokens* in the rest of the 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)

```
#define foo 17
```

```
void f() {
```

```
    int food = foo; // becomes int food = 17 (ok)
```

```
    int foo = 9+foo+foo; // becomes int 17 = 9+17+17 (nonsense)
```

```
}
```



## 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; } // my preference
```

Replace all matching “calls” with “body” but with *text of arguments* where the formals are.

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 a good idea to avoid performance overhead of a function call.

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,cnt) ((t*)malloc((cnt)*sizeof(t))
```

```
#define PRINT(x) printf("%s:%d %s\n",__FILE__,__LINE__,x)
```

# Conditional compilation

---

`#ifdef F00` (matching `#endif` later in file)

`#ifndef F00` (matching `#endif` later in file)

`#if F00 > 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 debugging-prints`  
`#define DBG_PRINT(x) printf("%s",x)`  
`#else`  
`#define DBG_PRINT(x) // replace with nothing`  
`#endif`

Note: `gcc -D F00` makes F00 “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 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` (cycles fine with “the trick”, the way you say what other definitions you need)
- `#define` (avoids magic constants, parameterized macros have a few justifiable uses, token-based text replacement)
- `#if...` (for showing the compiler less code)

## printf and scanf

---

“Just” two library functions in the standard library

- Prototypes in `stdio.h`

Example: `printf("%s: %d %g ", x, y+9, 3.0)`

They can take any number of arguments.

- You can define functions like that too, but it is rarely useful, arguments are not checked for any types, and writing the function definition is a pain.
  - Not covered in 303.

The `f` is for “format” – crazy characters in the format string control formatting.

## The rules

---

To avoid HYCSBWK:

- Number of arguments better match number of %
- Corresponding arguments better have the right types (%d,int %f,float, %e,float (prints scientific), %s,\0-terminated char\*, ... (look them up))

For scanf, arguments should be *pointers to* the right type of thing (reads input and assigns to the variables).

- So int\* for %d, but still char\* for %s (not char\*\*)

## More funny characters

---

Between the % and the letter (e.g., d) can be other things that control formatting (look them up; we all do).

- Padding (width) %12d %012d
- Precision ...
- Left/right justification ...

Know what is possible; know that other people's code may look funny.



## More on scanf

---

- Check for errors (returns number of % successfully matched)
  - maybe the input does not match the text
  - maybe some “number” in the input does not parse as a number
- Always bound your strings
  - Or some external data could lead to arbitrary behavior (common source of viruses; input a long string containing evil code)
  - Remember there must be room for the `\0`
  - `%s` reads up to the next whitespace

Example: `scanf("%d:%d:%d",&hour,&minutes,&seconds);`

Example: `scanf("%20s",buf)` (buf better have room for 20 characters)

## Useful, bizarre sublanguage

---

This is yet another funky little collection of characters with strange meaning.

- Pretty useful for reading/writing files (and the screen)
  - See `fprintf`, `fscanf`
- Also useful for reading/writing regular old strings
  - See `snprintf`, `sscanf`
  - (Do not use `sprintf` unless you enjoy danger.)