
CSE 374

Programming Concepts & Tools

Hal Perkins

Spring 2009

Lecture 10 – C: Preprocessor basics; printf/scanf

Agenda

- Two important “sublanguages” used a lot in C (almost every program)
 - The preprocessor: runs even before the compiler (hence the name)
 - Simple `#include` and `#define` for now; more later
 - `printf/scanf`: formatted I/O
 - Really just a library though
- We’ve seen these a little already

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 (now) and parameterized macros (textual-replacements) (later)
 3. Conditional compilation (later)

File inclusion

`#include <foo.h>`

- Search for file `foo.h` in “system include directories” (on fedora `/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 (use `gcc -E` if you want a look!)
 - 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 functions other files.
- `gcc -I dir1 -I dir2 ...` look in these directories for header files first (keeps paths out of your code files). We probably won't need to use this

Simple macros & symbolic constants

```
#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)
}
```

printf and scanf

- “Just” two library functions in the standard library
 - Prototypes in `stdio.h`
- Example: `printf("%s: %d %g\n", p, y+9, 3.0)`
- They can take any number of arguments
 - You can define functions like this too, but it is rarely useful, arguments are not checked for any types, and writing the function definition is a pain.
 - Not covered in 374
- 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 must 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**)
int n; char *s;
...
scanf(“%d %s”, &n, s);

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 (scanf 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)`

(better have room for ≥ 20 characters)