
CSE 374

Programming Concepts & Tools

Hal Perkins

Fall 2017

Lecture 8 – C: Miscellanea
Control, Declarations, Preprocessor, printf/scanf

The story so far...

- The low-level execution model of a process (one address space)
- Basics of C:
 - Language features: functions, pointers, arrays
 - Idioms: Array-lengths, strings as arrays with '\0' terminators
- Today – a collection of core C idioms/ideas:
 - Control Constructs, ints as booleans
 - Declarations & Definitions
 - Source file structure
 - Two important “sublanguages” used a lot in C
 - The preprocessor: runs even before the compiler
 - Simple #include and #define for now; more later
 - printf/scanf: formatted I/O
 - Really just a library though
- Next time: lvalues, rvalues, arrays & pointers; then structs & memory allocation

Control constructs

- while, if, for, break, continue, switch: much like Java
- Key difference: No built-in boolean type; use ints (or pointers)
 - Anything but 0 (or NULL) is “true”
 - 0 and NULL are “false”
 - C99 did add a bool library but not widely used (particularly in old code)
- goto much maligned, but makes sense for some tasks (more general than Java’s labeled break)
- Gotcha: switch cases fall-through unless there is an explicit transfer (typically a break), just like Java

Declarations and Definitions (1)

- C makes a careful distinction between these two
- *Declaration*: introduces a name and describes its properties (type, # parameters, etc), but does not create it
 - ex. Function prototype: `int twice(int x);`
 - also works (not as good style?): `int twice(int);`
- *Definition*: the actual thing itself
 - ex. Function implementation:
`int twice(int x) { return 2*x; }`

Declarations and Definitions (2)

- An item may be *declared* as many times as needed
 - although normally at most once per scope or file (i.e., can't declare the same name twice in a scope)
 - Declarations of shared things are often #included (read) from header files (e.g., `stdio.h`)
- An item must be *defined exactly once*
 - e.g., there must be a single definition of each function in only one file no matter how many files contain a declaration of it (or #include a declaration) or actually use it

Forward References

- No forward references allowed:
 - A function must be defined or declared in a source file before it is used. (Lying: “implicit declaration” warnings, return type assumed int, ...)
 - Linker error if something is used but not defined in some file somewhere (including main)
 - Use -c to compile file but not link to others (more later)
 - To write mutually recursive functions, you just need a (forward) declaration

Some (more) glitches

- Declarations must precede statements in a “block” in classic C
 - But any statement can be a block, so use { ... } if you need to
 - Or use -std=c11 (or c99) gcc compiler option to relax this restriction

Some (more) glitches

- Array variables in code must have a constant size
 - So the compiler knows how much space to allocate
 - (C99 has an extension to relax this – rarely used)
 - Arrays whose size depends on runtime information are allocated on the heap (next time)
 - Large arrays are best allocated on the heap also, even if constant size, although not required

More gotchas

- Declarations in C are funky:
 - You can put multiple declarations on one line, e.g.,
`int x, y;` or `int x=0, y;` or `int x, y=0;`, or ...
 - But `int *x, y;` means `int *x; int y;` – you usually mean (want) `int *x, *y;`
 - Common style rule: one declaration per line (clarity, safety, easier to place comments)

More gotchas

- Variables holding arrays have super-confusing (but convenient) rules...
 - Array types in function arguments are pointers(!)
 - Referring to an array doesn't mean what you think (!)
 - “implicit array promotion” (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 (now)
 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 Linux `/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 -P` if you want a look!)
 - Idea is simple: e.g., declaration for `fgets` is in `stdio.h` (use `man` for what header 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 (more later)
- `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” (tokens) 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)
}
```

Typical file layout

- Not a formal rule, but good conventional style

```
// includes for functions & types defined elsewhere
#include <stdio.h>
#include "localstuff.h"
```

```
// symbolic constants
#define MAGIC 42
```

```
// global variables (if any)
static int days_per_month[ ] = { 31, 28, 31, 30, ...};
```

```
// function prototypes (to handle "declare before use")
void some_later_function(char, int);
```

```
// function definitions
void do_this( ) { ... }
char * return_that(char s[ ], int n) { ... }
void some_later_function(char c, int n) { ... }
int main(int argc, char ** argv) { ... }
```

printf and scanf

- “Just” two library functions in the standard library
 - Prototypes (declarations) 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 usually not checked and writing the function definition is a pain
 - Writing these not covered in this course
- The “f” in `printf` 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))
 - Compiler might check, but not guaranteed
- 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);
```

*Hopefully You Crash Soon But Who Knows...

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)