Administrivia

• Homework 2 turned in

• Homework 3 out this afternoon: start early!
  • Ramp up again from HW2

• How’s everything going?
Where we are

• We’ve just set out to C

• Today we’re going to learn to navigate more of the language
  
  • Control Structures, Boolean Expressions, Null

  • Declarations and Definitions, Forward References, Array Declarations, Pointer Declarations

• C Preprocessor

• printf and scanf, convenient IO
Control in C

• if (<expr>) { <body> }

• if (<expr>) { <body> } else { <body> }

• while (<expr>) { <body> }

• do { <body> } while (<expr>)

• for (<init>; <expr>; <stmt>) { <body> }

• continue; break; switch…
Expressions

• There is no boolean type in C

• Instead, everything is true, except 0 and `Null`

• People have added their own boolean libraries, but nothing has stuck

• Same comparison operators as Java: `<`, `>`, `<=`, `>=`, `==`, `!=`

• You can also use a number as a boolean (or negate it with `!`)
Examples

• Loop 100 times
for (int i = 0; i < 100; i++) {
    printf("%d\n", i); //print out value of i
}

• Do something if x is not Null
if (x) {
    <something>
}
Null

• What is it? Nothing

• It’s the value stored in a pointer which points nowhere

• *NEVER* dereference Null

• Used to denote “nothing’s here”

• Think of it as a blank treasure map, leading nowhere
Declarations/Definitions

• Declaration
  • Telling the world something is there
  • Only concerned with external shape, or type
  • As many times as you want (but only once per file/scope)

• Definition
  • Filling in the internal bits
  • Only once
Functions

• Declaration

```c
int twice(int x);
```

• Definition

```c
int twice(int x) {
    return 2 * x;
}
```
Forward References

• Anything you use must be declared before use

• (Defining counts as declaring)

• **Cannot** have forward references:
  ```c
  int main(int argc, char* argv[]) {
    int y = twice(argc);
    printf("%d\n", y);
  }

  int twice(int x) {
    return 2 * x;
  }
  ```
How to structure a file

• If you have 2 functions f and g, and f calls g, define g before you define f

• If they need to call each other, you have to declare one before defining it
Classic C

- In old classing C, all variable declarations need to come at the beginning of a block

- Thankfully that is no longer the case

- Ignore your book on this one
Array Declarations

• Uninitialized Arrays:
  int n[10];
  char buffer[128];

• Initialized Arrays:
  int n[3] = {0,0,0};

• As Function Parameters:
  int sum(int x[], int x_length) { ... }

Actually a pointer
Multiple Declarations

• You can put multiple declarations on one line

  e.g. int x, y, z;

• This will get you in trouble fast!

  e.g. int* x, y, z;

• One declaration per line, especially if it’s a pointer type
C Preprocessor

• Rewrites your files before the compiler gets code
• Everything that starts with #
• This can do normal and crazy things
  • Please stick to the more normal
    1. Include header files (Today)
    2. Define constants (Today) and parameterized macros (Later)
    3. Conditional compilation (Later)
#include

- `#include <foo.h>`
  - Look for `foo.h` in “system directories”, find and preprocess contents (recursion), and paste results literally (as a string) into this file

- `#include "foo.h"
  - Same as above, but look in current directory first

- `gcc -I dir1 -I dir2` will pass in search directories for header files (we won’t need in this class)
Macros and Constants

- `#define` replaces `tokens` in the rest of the file

- Knows where words (tokens) start and end (unlike `sed`)

- No notion of scope
  
  ```c
  #define foo 17
  void f() {
    int food = foo;  // int food = 17;
    int foo = 9+foo+foo;  // int 17 = 9+17+17;
  }
  ```
printf and scanf

- “Just” two library functions
  - Declared in `<stdio.h>`
- Used to print to stdout and read from stdin
- They can take any number of arguments
- The “f” in name stands for formatted
printf and scanf

- Number of arguments better match number of % in format string
- Corresponding arguments better have the right type
  - For scanf must be pointer type (int* for %d, still char* for %s)
- Compiler probably won’t check for you
- If you don’t follow rules, hopefully you crash soon, but who knows?
printf and scanf

• Many different formatting options

• Read documentation to find all of it
  • Padding, precision, left/right, decimal/hex, etc…

• You **must** check scanf to see if it worked
  • input may not have matched text
  • maybe some number typed in not a number
scanf

• scanf looking for a string (%s) will read until whitespace, and write into provided string

• If you don’t have enough space, it will overwrite something else

• You can limit it with %20s or %45s

• The number given is number of characters, you still need more room for ‘\0’ terminator