C programming

C differences (from Java, etc):
- procedural
  o no classes
- In standard C must be declared on top of code block, before actual code.
- No new / delete
  o Use malloc/free
- No boolean types
  o 0 false, anything else true
- Typechecking not strictly enforced

Program components:
- Preprocessor:
  #include – preprocessor directive, inlines given file
  For standard input/output library functions and other useful functions:
    #include <stdio.h>
    #include <stdlib.h>

    #define X Y -- replaces all occurrences of X with Y. Use for declaring constants or macros. I.e. #define OP_j 0x02

- can directly declare global variables (unlike Java)
- functions and code declared just like Java.
- Variables work the same way: int, char, double, etc.
- Same loop constructs: for, while, do..while
- Casts… int x; char y; y = (char)x;
- Arrays: int a[10]; declares an array which holds 10 ints. Works for 2D, etc.
  o Zero-based
  o Can init like int a[10] = {0,1,2,…};
- Pointers: declare a variable which holds an address to some other variable.
  o I.e. int *a; declares a to hold a pointer to some integer. **does not** allocate the integer itself!
  o & takes address of a variable.
    § int b;
    § a = &b;
  o * dereferences a pointer, i.e. essentially executes a lw or a sw
    § *a = 42; // move $t0, 42; la $t1, a; sw $t0, 0($1)
    § b = *a; // load contents of a, move into b.
  o An array variable is actually a pointer to the first array element.
    § i.e. above, a is of type (int *) and points to start of the array in memory.
- char *str - strings are commonly represented and passed around like this, as an array of characters. No explicit length is stored! (buffer overruns…)

- Powerful, lead to many subtle bugs, difficult to debug. USE CAUTION!

- Probably won’t need in assignment except maybe for arrays or strings. Won’t need pointer arithmetic.

- Structs:
  - Not classes – they don’t have methods
  - struct struct_name {
      Int a,b,c;
      Char *str;
    } my_struct;
    my_struct.a = str;

    my_struct is an instantiation of struct which is named by struct_name. To declare another such struct, you can use:
    struct struct_name my_struct2;
    Both struct name and instantiation parts are optional. The instantiation part can be a comma-separated list.

- Need main function:
  int main(int argc, char **argv)

  argc = number of arguments,
  argv = array of arguments; each argument is a string, first argument is always program name
  Both are optional

Compiling:
- gcc –o myprg myprg.c [otherfile1.c otherfile2.c …]
- -o flag tells the output executable name
- -Wall prints out all warnings
- -O6 turns on full optimization (that’s O, not zero)
- -g enables debugging information to be used

Debugging:
- gdb myprg
- Look up commands

Reference:
- Issue “man printf” or any other function

Programming tips:
- Bit manipulation:
  - << and >> are your friends. >> will do arithmetic or logical shift depending on whether the operand is signed or not.
- Know & and | (and, or), and how to form bitmasks. Also, ^ is XOR, ~ is NOT, both bitwise. ! is boolean not.
- Extract bits 0..m in a 32-bit number: x & ((1 << m)-1)
- Extract bits m..31 in a 32-bit number: x >> m
- Multiply by 2^n shift left by n
- Divide by 2^n shift right by n (which shift, logical or arithmetic?)
- What does a power of 2 look like? (only one bit set)
- Example: given instr, how do you extract rd (bits 15…11):
  $ (instr >> 11) & 0x1f$
  $ 0x1f = 11111$
- Assignment shortcuts (sometimes confusing, should understand):
  - Avoid replicating variable when used as both source and destination
  - x++, x+=n, x &= 0xff, etc.
  - x = x >> 1 \ x >>= 1;
    *str = ’z’; str++;    *str++ = ’z’;
- Can use booleans in statements:
  if ( x == y ) return 1; else return 0;   return x == y;
- Output:
  - printf(format string, values); (variable length)
  - Examples:
    $ printf(“What I want to print…
”);
    $ printf(“My decimal number: %d
”, num);
    $ printf(“My number printed as hex: %x, with big digits: %X
”,
      num, num);
    $ char str[256] = “this is a test!”;
      printf(“My string: %s
”, str);
  - End your strings with \n, newline
- Strings:
  - Must preallocate space for them!
  - Cannot declare char *buf and then use buf.
  - Either preallocate statically, or use malloc:
    $ char buf[1024] allocates 1024 characters;
  - Can initialize when declaring:
    - Char buf[1024] = “Hello!”;
    - Will put six characters into buf (hello and !), and also put
      the null termination character. Will not clear the rest.