CSC369 Tutorial 1

Some review material
How to succeed in this course

• Show up to lectures & tutorials
  – More material to cover than lecture time available
• Work on assignments evenly and collaborate
  – “Fill your partners in” and make sure you all understand *everything*.
• Compiler warnings!
  – In the past, automatic 10% penalty on assignments.
• SVN
  – `svn add`; do a clean checkout and build (from scratch) before you submit your assignments
How to succeed in this course

• Read assignments carefully; lots of corner cases & design decisions to make
• Read the documentation
• Keep things modular
  – Make this part of your initial design
• Use the tools available to you & be proactive in learning them
  – Good for industry as well
• Design documents
  – More than line-by-line descriptions of your code
• Explain the design (how/why); don’t regurgitate the code
C REVIEW
Some C Review!

• Go through these slides (and try the exercises…) at home!
• Brush up / learn what you don’t know now!
  – Assignments are work-intensive enough as it is…
• Topics: Bit manipulations, pointers, argument-passing, arrays, pointer arithmetic, memory allocation, error handling, etc.
Pointers

• Every variable has a memory address
  – Can be accessed with “address of” operator: &

• Pointers are variables that store memory addresses
  – int x = 42;
  – int *x_ptr = &x;
  – int *heap_ptr = (int *)malloc(sizeof(int));

• The value a pointer refers to can be accessed with *
  – This is “dereferencing”
    • int y = *x_ptr;
NULL

• NULL is the “0” value for addresses.
  – It’s a good idea to initialize pointers to NULL.
    • Including fields of structures
    • Check pointers for NULL before dereferencing
    • Much easier to catch bugs!
  – It’s often used as an error value, too.
Pass by Value / Reference

• C only allows one value (which may be a struct) to be returned.

• If variables are passed into a function by value, any changes to them will not be seen outside the function.
  – Why? A copy of each parameter is made on the stack, and changes are made to the copy.

• If pointers are passed into a function, any changes made to the values they point to will be seen -- this is passing by reference.
  – Note that the pointers themselves are still passed by value!

ex1.c
Arrays

• Arrays contain multiple variables of the same type.
• Each element can be accessed with [] notation.
  
  ```
  int x_arr[10];
  for (i = 0; i < 10; i=i+1)
    x_arr[i] = i;
  ```

• Arrays are ... almost the same as pointers.
  After “int *x_ptr = x_arr;” x_ptr[i] is just like x_arr[i]
  – Differences:
    • sizeof(x_ptr) = 4 (sizeof(int*)), whereas sizeof(x_arr) = 40 (10*sizeof(int))
    • You can’t change an array var. to point to a different array
  – Note: arrays are passed to a function as a **pointer**, not an array-typed variable
Pointer Arithmetic

• Pointers are just values, so you can manipulate them.
• If x is an array, this is true:
  x[5] == *(x + 5)
• The key? Constants added to pointers are “scaled” by the size of the type. Adding 5 to an (int *) adds 5 * sizeof(int).
Pointers and Structs

• Structs are one “aggregate” structure in C.
  – A struct can contain multiple variables in a single
    – package.
• Structs have a syntactic quirk:
  – If you have a struct variable, use “.”
    struct mystruct s= ...
    s.myfield = 6;
  – If you have a struct pointer, use “->”
    struct mystruct *s_ptr = ...
    s_ptr->myfield = 6;
    (*s_ptr).myfield = 6;
Allocating Memory

• malloc allocates memory from the heap
  – It allocates by byte, so it requires a size
  – Its return value must be typecast
    int *heap_ptr = (int *)malloc(sizeof(int) * 4);
• Don’t forget to “free” memory you “malloc”!
• Remember to use “kernel” versions of the calls if you’re working inside the kernel
  – Instead of malloc, kmalloc
  – Instead of free, kfree
Stack Allocation

- Heap allocation isn’t always necessary
- Also might cause a memory leak (if not careful...)

```c
int foo() {
    struct mystruct z;
    z.x = 1;
    return funcwithmystruct(&z);
} ..... NOT

int foo() {
    struct mystruct* z = malloc(sizeof(struct mystruct));
    int rval = -1;
    z->x = 1;
    rval = funcwithmystruct(z);
    free(z);
    return rval;
}
```
Stack versus Heap trade-off

• Stack allocation is “easy,” but stack sizes are limited. (1-4MB for a “regular” system, and only 4KB for a kernel thread running on sys161)
  – This means any array or struct with more than a handful of elements should be heap allocated.
  – Cannot return a pointer to a stack-allocated variable
  – Additionally, no recursion in kernel threads!

• Heap allocation is “harder,” but gets around these limitations. Why is it harder?
  – Have to remember to free any malloc’d mem.!
  – Can’t free a memory location more than once!
Don’t Leak Memory!

• Make sure to free memory you allocate
• This example shows an error case

```c
struct mystruct* sys_mystruct() {
    struct mystruct* first;
    first = malloc(sizeof(struct mystruct));
    if (first == NULL) {
        return -1;
    }
    first.other = malloc(sizeof(struct otherstruct));
    if (first->other == NULL) {
        return -1;
    }
    return first;
}
```
More C Quirks to Remember

• Uninitialized variables
  – … have undetermined value (and C won’t complain)
• Array bounds
• Runtime exceptions
  – … don’t exist!
  – Instead, functions return, e.g., “-1” or “0”
• Type casts
  – … are not checked at runtime! (can cast char to int*)
  – “Dangerous,” but you’ll need to do it sometimes.
• Memory can be corrupted without the program crashing: check your bounds!
• Use assert() to check invariants
  – but not error conditions that are actually possible!
C Error Messages

• Segmentation Fault:
  – A pointer has accessed a location in memory that is not in a segment you own.
  – Maybe an infinite loop: overran an array?
  – Forgot to initialize a pointer and dereferenced it?
  – Adding two pointers that shouldn’t be?
  – Note: segfaults can be sporadic, since you have to step outside the (rather large) segment to get one.

• Bus Error:
  – A pointer is not properly aligned.
  – Bad casting? Bad pointer arithmetic?
General Tips

• Simplify whenever possible
  ```
  struct mystruct myarray[10][10];
  ```
  is better than
  ```
  struct mystruct **myarray;
  ```
• Declare all functions ahead of time
• Use a test-oriented **incremental** development strategy
  – Test first and frequently
C: bit manipulation

• Sometimes we need to alter bits in a byte or word of memory directly
  – A 32-bit int is a very compact way to represent 32 different boolean values

• C provides bitwise boolean operators
  – “&” : AND
  – “|” : OR
  – “~” : NOT (complement)
  – “^” : XOR (exclusive OR)
## Practice with bit ops

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>0110 1001</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>0101 0101</td>
</tr>
<tr>
<td>~a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>~b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &amp; b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a ^ b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Practice with bit ops

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0110 1001</td>
</tr>
<tr>
<td>b</td>
<td>0101 0101</td>
</tr>
<tr>
<td>~a</td>
<td>1001 0110</td>
</tr>
<tr>
<td>~b</td>
<td>1010 1010</td>
</tr>
<tr>
<td>a &amp; b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
## Practice with bit ops

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0110 1001</td>
</tr>
<tr>
<td>b</td>
<td>0101 0101</td>
</tr>
<tr>
<td>~a</td>
<td>1001 0110</td>
</tr>
<tr>
<td>~b</td>
<td>1010 1010</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>0100 0001</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a ^ b</td>
<td></td>
</tr>
</tbody>
</table>
## Practice with bit ops

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0110 1001</td>
</tr>
<tr>
<td>b</td>
<td>0101 0101</td>
</tr>
<tr>
<td>~a</td>
<td>1001 0110</td>
</tr>
<tr>
<td>~b</td>
<td>1010 1010</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>0100 0001</td>
</tr>
<tr>
<td>a</td>
<td>0111 1101</td>
</tr>
<tr>
<td>a ^ b</td>
<td>0111 1101</td>
</tr>
</tbody>
</table>
## Practice with bit ops

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0110 1001</td>
</tr>
<tr>
<td>b</td>
<td>0101 0101</td>
</tr>
<tr>
<td>~a</td>
<td>1001 0110</td>
</tr>
<tr>
<td>~b</td>
<td>1010 1010</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>0100 0001</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a ^ b</td>
<td>0011 1100</td>
</tr>
</tbody>
</table>
Bit Shifting

- \( x << k \) : shift the bits of \( x \) by \( k \) bits to the left, dropping the \( k \) most significant bits and filling the rightmost (least significant) \( k \) bits with 0

- Example: \( 6 << 1 = 12 \)
  Before: 00000000 00000000 00000000 00000110
  After: 00000000 00000000 00000000 00001100

Equivalent to multiplying by \( 2^k \)
Bit Shifting

- Shifting is *non circular*
- E.g 3,758,096,384 $\ll 1$

- Before: 111000000 00000000 00000000 00000000
- After: 110000000 00000000 00000000 00000000

- What if $k$ is $\geq$ size of object? (e.g., for int’s, on 32-bit machine, $k \geq 32$)
  - UNDEFINED! Don’t assume the result will be 0
Bit Shifting

• $x \gg k$ right shift, logical or arithmetic
  • logical right shift - fill left end with $k$ 0’s (unsigned types)
  • arithmetic right shift (care about signed bit) - fill left end with $k$ copies of the most significant bit
    • C does not define when arithmetic shifts are used! Typically used for signed data, but not portable

• Example -2,147,483,552 $\gg$ 4

• Before: 10000000 00000000 00000000 01100000
• Arithm: 11111000 00000000 00000000 00000110
• Logical: 00001000 00000000 00000000 00000110
Bit Masks

- A mask is a bit pattern that indicates a set of bits in a word
  - E.g., 0xFF would represent the least significant byte of a word
  - For a mask of all 1’s, the best way is ~0
    - Portable, not dependent on word size
    - For 32-bit machines, 0xFFFFFFFF will work
    - You may also see -1 used (2’s complement, -1 is a bit pattern with all bits set to 1)
Practice with bit masks

• Given an integer \( x \), write C expressions for:
  – Set \( n \)-th bit of \( x \):
Practice with bit masks

• Given an integer x, write C expressions for:
  – Set n-th bit of x:
  • int y |= 1 << n
  – L.s.byte unchanged, toggle all other bits of y:
Practice with bit masks

• Given an integer x, write C expressions for:
  – Set n-th bit of x:
    • int y |= 1 << n
  – L.s.byte unchanged, toggle all other bits of y:
    • int y ^= 0xffffffff00
Practice with bit masks

• Given an integer x, write C expressions for:
  – Least significant byte of x, all other bits set to 1:
    • int y = _______________________________________________________________________
  
  – Complement of the l.s.b. of x, all other bytes unchanged:
    • int y = _______________________________________________________________________

  – All but l.s.b. of x, with l.s.b. set to 0
    • int y = _______________________________________________________________________
Practice with bit masks

• Given an integer x, write C expressions for:
  – Least significant byte of x, all other bits set to 1:
    • int y = x | 0xFFFFFFFF00

  – Complement of the l.s.b. of x, all other bytes unchanged:
    • int y = ________________________________

  – All but l.s.b. of x, with l.s.b. set to 0
    • int y = ________________________________
Practice with bit masks

• Given an integer x, write C expressions for:
  – Least significant byte of x, all other bits set to 1:
    • int y = x | 0xFFFFFFFF00
  – Complement of the l.s.b. of x, all other bytes unchanged:
    • int y = x ^ 0xFF
  – All but l.s.b. of x, with l.s.b. set to 0
    • int y = ________________________________
Practice with bit masks

• Given an integer x, write C expressions for:
  – Least significant byte of x, all other bits set to 1:
    • int y = x | 0xFFFFFFFF00
  – Complement of the l.s.b. of x, all other bytes unchanged:
    • int y = x ^ 0xFF
  – All but l.s.b. of x, with l.s.b. set to 0
    • int y = x & 0xFFFFFFFF00
Exercise 1

In groups (max 3)