Purpose: More advanced C programming.

Details: We’re taking a little break from assembly language to review and refresh our knowledge of C and to explore a few more advanced concepts. Today we look at arrays and pointers, and also at bit manipulation.

Recall (from the class notes of October 1) that a pointer variable holds an address of a location in memory. Since different kinds of data require different amounts of memory, the declaration of a pointer variable includes a description of the kind of data that is being pointed to. For instance:

```c
int *iptr; /* declares a pointer to integer */
double *dptr; /* declares a pointer to double */
char *cptr; /* declares a pointer to char */
```

When we declare an array, such as “int a[10];”, we are actually declaring a pointer variable named a that contains the address of the first element in the array. If we want to know the value stored there, we must dereference it with the “*” operator: *iptr is the integer stored at location iptr.

We can add or subtract values from pointer variables, just as we have seen (in MIPS) that we can add or subtract values from addresses in assembly language. Because it knows the type of value pointed to, the C compiler translates constants into the appropriate number of bytes for that type of data. For instance,

```c
iptr = iptr + 1; /* adds 4 to the address in iptr */
dptr = dptr - 1; /* subtracts 8 from the address in dptr */
cptr = cptr + 3; /* adds 3 to the address in cptr */
```

1. [Modify the sample program.] In the course shared repository, in folder “lab7”, study the program “pointer.c”.

   (a) Write a program named dpointer.c that does exactly the same thing as program pointer.c, only with an array of double values rather than int values. (You will need to change several things to get this to work, e.g., format specifiers, etc.) Be sure to choose a pleasing output format so that lines don’t wrap around the edge of the screen.

   (b) Write a program named llpointer.c that does exactly the same thing as program pointer.c, only with an array of long long values rather than int values. The format for printing a long long integer is “%lld”.

Handed out on 19 October 2015
(c) [Easy bonus problem.] How many bytes are needed for a long long integer? Write a little C program that prints out how many bytes are needed for an int, a float, a double, a char, a short, a long, and a long long variable. (HUGE HINT: C has an operator named sizeof—see page 135 in K&R.)

2. [Masking bits.] Study program “chars.c” in the lab7 folder of the class repository.

Now write a C program that asks the user to input three integers $a$, $\begin{array}{c} \text{begin} \end{array}$, and $\begin{array}{c} \text{end} \end{array}$. The value $a$ should be saved in an unsigned int. The values $\begin{array}{c} \text{begin} \end{array}$ and $\begin{array}{c} \text{end} \end{array}$ should be integers in the range 0 to 31 and should specify starting and ending bit positions in $a$.

The program should print out the integer formed by just the bits in positions $\begin{array}{c} \text{begin} \end{array}$ through $\begin{array}{c} \text{end} \end{array}$ of the integer $a$.

For instance:

```c
$ ./a.out
Enter an unsigned integer $a$: 127
Enter beginning and ending bit positions (between 0 and 31): 25 31
Answer = 127
$ ./a.out
Enter an unsigned integer $a$: 127
Enter beginning and ending bit positions (between 0 and 31): 25 25
Answer = 1
$ ./a.out
Enter an unsigned integer $a$: 127
Enter beginning and ending bit positions (between 0 and 31): 25 26
Answer = 3
$ ./a.out
Enter an unsigned integer $a$: 127
Enter beginning and ending bit positions (between 0 and 31): 0 30
Answer = 63
$ ./a.out
Enter an unsigned integer $a$: 2147483648
Enter beginning and ending bit positions (between 0 and 31): 0 2
Answer = 4
```

SUGGESTION: You need to create a mask based upon the values of $\begin{array}{c} \text{begin} \end{array}$ and $\begin{array}{c} \text{end} \end{array}$. If you start with all 1s (i.e., $\text{mask} = 0xFFFFFFFF$), you can shift $\text{mask}$ left by $\begin{array}{c} \text{begin} \end{array}$ bits, then shift right by $\begin{array}{c} \text{begin} \end{array}$ bits. That strips off the leading 1s before position $\begin{array}{c} \text{begin} \end{array}$. A similar trick lets you strip off the 1s after position $\begin{array}{c} \text{end} \end{array}$ (do a right shift, then a left). Now you have a mask with 1s only in positions $\begin{array}{c} \text{begin} \end{array}$ through $\begin{array}{c} \text{end} \end{array}$

Submit the fully commented C programs by 8 a.m. on October 26.