CS 237 Meeting 9 — 9/26/12

Announcements

1. None?

Arrays and Pointers in C
Simple pointer parameters and variables

- Recall the swap method we considered during one of our first few classes.

```
#include <stdio.h>

// A swap that does not work.
void swap( int x, int y ) {
  int temp;

  temp = x;
  x = y;
  y = temp;
}

int main( int argc, char * argv[] ) {

  int a = 10;
  int b = 100;

  swap( a, b );

  printf( "a = %d, b = %d\n", a, b );

}
```

Figure 1: incorrect version of swap.c

- That method did not work because when you pass a parameter in C, you only pass the value of the parameter. Even if the actual parameter is a variable, all that the called function gets to access is its value. The called function cannot modify the actual parameter.
- We used this example to explain why you needed an ampersand before the parameters you pass to scanf. The ampersand tells C to pass a *pointer* to the actual parameter, a value that describes the location where the variable can be found in memory, rather than the value of the actual parameter.
- Now, we want to go a bit further and see how a swap method that works can be written using pointers.
- All we have to do is:
  - Put asterisks in front of each of the two parameter names in their declarations in swap.

    Including an asterisk like this in any declaration tells C that the variable will hold a pointer to something of the type that would have been stored in the variable if the asterisk were not present.
  - Put asterisks in front of each use of the formal parameter names in swap.

    Putting an asterisk before the use of a variable in an expression only works if that variable is a pointer. If so, it tells C that the programmer wants to access the variable that the pointer points to rather than the pointer itself. This is called *derferencing*.
    Note that the asterisk has completely different meanings depending on whether it is used as a infix or prefix operator.
  - Put ampersands in front of the parameters when swap is called.
  - The improved version is shown in Figure **??**.
- If we run the program with a breakpoint, we can actually print the pointer values. They are displayed in hexadecimal. If this throws you, we can force the debugger to print them as decimal numbers by adding "(int)" after the print command name.
  - Note that the two pointers differ by 4. It takes 4 bytes to store an integer on an x86 machine. Memory is basically a big array of

```
#include <stdio.h>

// A swap that works!
void swap( int * x, int * y ) {
  int temp;

  temp = *x;
  *x = *y;
  *y = temp;
}

int main( int argc, char * argv[] ) {

  int a = 10;
  int b = 100;

  swap( &a, &b );

  printf( "a = %d, b = %d\n", a, b );

}
```

Figure 2: Working version of swap using pointers

```
#include <stdio.h>

// Print all of the commands arguments on separate lines
int main( int argc, char *argv[] ) {

  int p;
  for ( p = 0; p < argc; p++ ) {
    printf( "Argument %d = %s\n", p, argv[p] );
  }

}
```

Figure 3: Program to echo arguments

bytes. Just as the registers in a machine are assigned numbers (0-31 for the MIPS), the elements (bytes?) of a machine's memory are assigned numeric addresses. Larger values (ints) are stored in a sequence of adjacent bytes. The value stored in a pointer is usually just the address of the first byte of the sequence used to store the value pointed to.

The argv array — An Introduction to arrays

- We have not seen how to use arrays yet in C, but we have seen one. Every main function you have written has a parameter declared as "char * argv[]". The parameter argv is an array of strings in C, just as it is in Java. But there are a few differences.
- The first one is in Java, we can ask an array like argv for its length. This is not possible in C. This is why main in C requires the int parameter argc to know how big argv is. In most C programs that use arrays, you will need separate int variables to keep track of the number of items in many arrays.
- Let's use this array to write a C program (shown in Figure **??**) that prints out its arguments on separate lines (we would print them on the same line, but the standard echo program already does that).

Declaring an array

- When you declare an array in C, you must provide its size. There is no separate "new int[x]" operation like there is in Java that creates/sizes the array. The allocation of the array is a part of its declarations (at least if it is declared as a global or local variable — we will see that array formal parameter declarations are different).
- The program member.c uses an array of ints to remember the integer values of its parameters and lets a user then check whether numbers are members of the parameter list.
- This program does strange things if you type in too many argument values. The array is declared to hold 10 values. If you enter 11, the program stores the 11th in values[10] and C does not detect this as an error (Java would have!). Instead, the 11th value ends up in the memory word right after values[9]. This memory word could be used for some other purpose leading to evil problems.

Passing an Array as a parameter

- Global variables are discouraged as a matter of good programming style.
- We could, therefore, improve the member.c program by moving the declaration of values into main. This makes it a local variable but requires that we pass it as a parameter to inList. A version of inList updated to make this possible is shown in Figure **??**.
- Note that you do not specify the size of the array in a formal parameter declaration. Typically, however, the receiving function will require an int parameter that tells it the size of the array.

Simple Sorting

- Figure **??** shows C code for a simple implementation of bubble sort.
- This code uses the swap function shown in Figure **??** in an interesting way. It always passes it a pair of consecutive elements of the values array.

```
#include <stdio.h>
#include "bools.h"

// The values from the command line to be searched
int values[10];

// The number of values that were entered
int valc;

// Determine whether the parameter value can be found in values
int inList( int val ) {
  int p;

  for ( p = 0; p < valc; p++ ) {
    if ( values[p] == val ) {
      return TRUE;
    }
  }
  return FALSE;
}

int main( int argc, char *argv[] ) {

  // Convert command arguments into an int array
  int p;
  valc = argc-1;
  for ( p = 0; p < valc; p++ ) {
    values[p] = atoi( argv[p+1] );
  }

  // Repeatedly search for numbers entered by the user
  printf("Enter a number: " );
  int searchNum;
  scanf( "%d", &searchNum );
  while ( searchNum > 0 ) {
    if ( inList( searchNum ) ) {
      printf( "%d is in the list\n", searchNum );
    } else {
      printf( "%d is not in the list\n", searchNum );
    }

    printf("Enter a number: " );
    scanf( "%d", &searchNum );

  }
}
```

```
int inList( int val, int valueList[], int valc ) {
  int p;

  for ( p = 0; p < valc; p++ ) {
    if ( valueList[p] == val ) {
      return TRUE;
    }
  }
  return FALSE;
}
```

Figure 5: inList function with an array parameter

```
  int outOfOrder = TRUE;

  while ( outOfOrder ) {
    outOfOrder = FALSE;
    for ( p = 0; p < valc - 1; p++ ) {
      if ( values[p] > values[p+1] ) {
            outOfOrder = TRUE;
            swap( &values[p], &values[p+1] );
      }
    }
  }
```

Figure 6: Bubble Sort

- Arrays are stored in memory in such a way that consecutive elements are stored in memory locations with "consecutive" addresses. Consecutive is quoted because if our array contains values that take 4 bytes of memory (as integers do), then the addresses of these consecutive words will differ by 4 since memory is usually byte-addressable.
- Oddly, if we use the GDB debugger to calculate the difference between two pointer values passed to swap by the bubble sort function while executing the codes of swap by typing

  (gdb) print y - x

  the answer displayed will be 1 rather than 4.
- There is a good reason for this. When C does arithmetic with pointers, it assumes the programmer wants to think abstractly about the things the pointers point to rather than about memory addresses. Therefore, if two pointers refer to items in consecutive locations in an array, their difference should be 1 even if the actual addresses for the array elements differ by 4.
- To pull this off:
  - The difference between two pointer values (that point to the same type) will always be the difference of the memory addresses involved divided by the number of bytes required to store a value of the type begin pointed to.
  - When we add/subtract an int constant from a pointer, the constant is first multiplied by the size of the item the pointer refers to.

Pointers and Array indexing

- In C, an array name by itself produces the address of the 0th element of the array.
- As a result of this, and the rules explained above, if a is an array and x is an int, then a + x is the address of array element a[x] and therefore *(a+x) is equivalent to a[x].