Computer Science 50
Fall 2009
Scribe Notes

Week 7 Monday: October 19, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 7 Monday: October 19, 2009
Andrew Sellergren

1 Geek Humor (0:00–2:00)

- Now that Quiz 0 is behind us, you're all qualified to laugh at this.
- Since this week's problem set involves digital photography forensics, we thought this lolcat was particularly appropriate.

2 Announcements (2:00–15:00)

- This is CS 50.
- 1 new handout.
- Problem Set 5 has been released and tasks you with recovering a series of photos from a photo card dump file. To add to the fun, these photos, which depict places, people, and things on campus, will become a scavenger hunt that you and your section can complete in the hopes of winning a fantastic prize![1]
- We're soliciting beta testers for the CS 50 Cloud! The cloud is a cluster of servers and hardware which we purchased and configured over the summer. Eventually, we'll migrate all students off of NICE and onto the Cloud. One advantage of this will be to allow us to better serve your software needs come final project time, particularly in the case that your project requires some language or infrastructure that NICE doesn't natively support.
- Speaking of final projects, now is a great time to start thinking about them! Even if you have no idea *how* you might implement it, you should begin thinking about *what* you will implement. To help with the how, check out the numerous seminars which will be led by our teaching fellows. If you want to develop an iPhone app, for example, there's a seminar for that.[2] And not to worry, the usual Apple fees will be waived since we have a CS 50 course account. If you're interested in developing a smartphone app, you should express interest in obtaining one of the Android phones which have been generously donated by Google. To develop for the Android, you'll be using an *application programming interface* (API). You can think of this like a header file in C: an API defines a set of functions which allow you to interact with a library of code. We've assembled a list of Fun APIs for you to browse!
- Check out HarvardTweets! David himself is an avid tweeter. Twitterer. Twitnik? Thanks to Twitter's API, David spent very little time getting HarvardTweets, a site which aggregates tweets from Harvard users, up and running. On top of that, he was able to dig up a very cool spherical visualization of hash tags.

[1] Prize may not actually be fantastic.
[2] Get it?! It's like the commercials!

- The goal of the final project is, of course, for you to amaze yourself and us with what you've learned over the course of the semester. To facilitate this amazement, you'll have the opportunity to display your final project at the CS 50 Fair in December! More details on that to come.
- Dinner this Wednesday with David and Prof. Radhika Nagpal, who teaches a course on artificial intelligence (RoboSoccer!), will be hosted at 5:30 PM by Mather House.

3 Another Debugging Tool (15:00–25:00)

- Thankfully, there exists another tool besides GDB and the bulletin board for helping to debug your code. It's called Valgrind and its primary purpose is to check for memory leaks and errors in your programs. Let's try using it on a sample program called `memory.c`:

```
/****************************************************************************
 * memory.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demonstrates memory-related errors.
 *
 * problem 1: heap block overrun
 * problem 2: memory leak -- x not freed
 *
 * Adapted from
 * http://valgrind.org/docs/manual/quick-start.html#quick-start.prepare.
 ****************************************************************************/

#include <stdlib.h>


void f()
{
    int *x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int
main(int argc, char *argv[])
{
    f();
    return 0;
}
```

As you may have already noticed, one of the bugs in this program is that it fails to free the 40 bytes of memory that it allocates. Although the operating system will generally take care of this when the program exits, it shouldn't be relied upon to do so. Another noticeable bug is the accessing of the 11th element in the `x` array even though we've only allocated enough memory for 10 elements. Remember that arrays are zero-indexed. This second bug won't necessarily cause a segfault. Using Valgrind, however, we can track down bugs like this that might otherwise lay in hiding.

- Once we compile `memory`, we'll run the command `valgrind memory`. We'll get a lot of cryptic-looking output, but toward the bottom we'll see the following line:

```
==26082== LEAK SUMMARY:
==26082==    definitely lost: 40 bytes in 1 blocks.
```

At the bottom also appears a suggestion to rerun the program using the configuration option `--leak-check=full`. In Linux, configuration options, which tweak the behavior of the program, begin with dashes by convention. Let's take its advice and reexamine the output:

```
==26210== Invalid write of size 4
==26210==    at 0x804853F: f (memory.c:22)
==26210==    by 0x804855C: main (memory.c:29)
==26210==  Address 0x41A6050 is 0 bytes after a block of size 40 alloc'd
==26210==    at 0x4023595: malloc (vg_replace_malloc.c:149)
==26210==    by 0x8048535: f (memory.c:21)
==26210==    by 0x804855C: main (memory.c:29)
==26210==
==26210== ERROR SUMMARY: 14 errors from 7 contexts (suppressed: 0 from 0)
==26210== malloc/free: in use at exit: 40 bytes in 1 blocks.
==26210== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.
==26210== For counts of detected errors, rerun with: -v
==26210== searching for pointers to 1 not-freed blocks.
==26210== checked 76,076 bytes.
==26210==
==26210==
==26210== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==26210==    at 0x4023595: malloc (vg_replace_malloc.c:149)
==26210==    by 0x8048535: f (memory.c:21)
==26210==    by 0x804855C: main (memory.c:29)
==26210==
==26210== LEAK SUMMARY:
==26210==    definitely lost: 40 bytes in 1 blocks.
==26210==      possibly lost: 0 bytes in 0 blocks.
==26210==    still reachable: 0 bytes in 0 blocks.
==26210==         suppressed: 0 bytes in 0 blocks.
```

Actually, this lengthy output isn't even everything—we just skipped to the part that pertains to our code. The message "Invalid write of size 4" which points to line 22 of our program seems to suggest that we're improperly assigning the value of an `int`. We might guess that this corresponds to our accessing past the bounds of the `x` array. The next error seems to point to our allocation of 40 bytes of memory which subsequently lost.

- Let's try to fix these errors. If we change our array assignment to be `x[9] = 0;`, then the first error referring to "Invalid write" disappears. Now if we add the line `free(x)` at the end of `f`, the other error disappears as well. What's interesting is that even after we've fixed these two errors, we get an error count of 13. Realize that anytime you compile a C program, you're most likely linking in libraries of code written by other people. These libraries aren't necessarily perfect, but neither are they buggy, as this run of Valgrind might suggest. Valgrind is actually extraordinarily anal, so if it finds errors in library code, don't immediate start wagging your finger. The most important message we get from Valgrind is as follows:

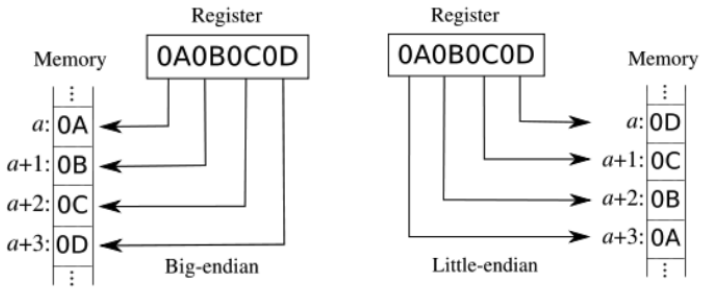
```
==26913== All heap blocks were freed -- no leaks are possible.
```

- You might be dismayed if you go back and run Valgrind on your code from Problem Set 4. If you used pointers at all, chances are you made at least one mistake that Valgrind will happily catch. Valgrind will be especially important to you as you implement Problem Set 6, our last problem set in C, which tasks you with building the fastest spellchecker possible.

4 Hexadecimal and Endianness (25:00–35:00)

- This week's problem set will expose you to examining hexadecimal values, which, as we've already seen, are used to represent colors in pixels. You'll be using a program called `xxd` which will dump the binary contents of a file in hexadecimal format.

- The format in which data is stored is largely operating-system- and CPU-specific. There exists the notion of *endianness*, which refers to the order in which bytes are written on disk. The number 1 in hexadecimal, for example, might be written in human-readable format as 0x00000001, but on NICE, which is a little-endian CPU, it would be represented as 0x01000000. Little-endian means that the lowest-order or least-significant byte,—that is, the byte that represents the smallest number—which, in this case is 0x01, is the leftmost byte. This order ultimately corresponds to how the values are laid out in memory, as the following diagram shows:



On a big-endian processor, the order in which bytes are stored in registers is the same as the order in which they are stored in memory. In other words, the least-significant byte is stored in the lowest memory address. The opposite is true of little-endian processors. Although processors don't agree on endianness, there at least exists a standard for endianness in the world of networking protocols.

- For Problem Set 5, you'll find that the values of the RGB triples which make up a bitmap are actually displayed in little-endian order when you examine them with `xxd`. Take a look at `endian.c`, which introduces you to the process of opening and manipulating the bytes of a bitmap file:

```
/**************************************************************************
 * endian.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Reads bf.bfSize from a BMP.
 *
 * Demonstrates endianness.
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>


int
main(int argc, char *argv[])
{
    // ensure proper usage
    if (argc != 2)
        return 1;

    // open file
```

```
FILE *fp = fopen(argv[1], "r");
if (fp == NULL)
    return 1;

// seek to BITMAPFILEHEADER's bfSize
fseek(fp, 2, SEEK_SET);

// read in BITMAPFILEHEADER's bfSize
unsigned long bfSize;
fread(&bfSize, sizeof(bfSize), 1, fp);

// print bfSize
printf("\nbfSize: %ld\n\n", bfSize);

// return to start of file
rewind(fp);

// read in BITMAPFILEHEADER's raw bytes
unsigned char *buffer = malloc(14);
fread(buffer, 1, 14, fp);

// print field via cast
printf("bfSize: %ld\n\n", *((unsigned long *) (buffer + 2)));

// print individual bytes in decimal
printf("bfSize:   %d   %d   %d   %d\n",
       buffer[2], buffer[3], buffer[4], buffer[5]);

// print individual bytes in hexadecimal
printf("bfSize: 0x%x 0x%x 0x%x 0x%x\n",
       buffer[2], buffer[3], buffer[4], buffer[5]);

// print individual bytes in binary
printf("bfSize: ");
for (int i = 2; i < 6; i++)
{
    for (int j = 7; j >= 0; j--)
    {
        int mask = 1 << j;
        if (buffer[i] & mask)
            printf("1");
        else
            printf("0");
    }
}
printf("\n\n");
```

```
    // that's all folks
    return 0;
}
```

First we're doing a sanity check to make sure the user has provided a single command-line argument, which will be the name of a bitmap file. Then we'll try to open that file and move forward two bytes using the `fseek()` function, which resembles the fast forward button on a DVD player. Next, we read in to a variable called `bfSize`, which, because we've studied the specification of the BMP file format (included in the Problem Set 5 specification), we know represents how large the bitmap and is written in the first few bytes of the bitmap's header. Once we've gotten this value, we do a few tedious manipulations to alter how that value is displayed. The result of running `endian` on a bitmap file consisting of a single dot, is the following:

```
bfSize: 58

bfSize: 58

bfSize:   58   0   0   0
bfSize: 0x3a 0x0 0x0 0x0
bfSize: 00111010000000000000000000000000
```

All this program really demonstrates is that `bfSize` is stored in little-endian format, the 58, or 0x3a in hexadecimal, being written in the left-most place.

5 Bitwise Operators (35:00–67:00)

5.1 Examples and Notation

- Unlike arithmetic operators, like `+` and `-`, which operate on the whole of two variables, bitwise operators line up two variables and operate on them one bit at a time.
- Let's take the `&`, or bitwise AND, operator first. Given two numbers x and y, x `&` y is the result of taking the corresponding bits of x and y and combining them using AND. As we've already seen with conditions in C, AND returns true (or 1) when both operands are true (or 1). If x = 0000 and y = 0001, then x `&` y = 0000 because, from left to right, the first three bits are equal to 0 `&` 0, or 0, and the fourth bit is equal to 0 `&` 1, or 0.
- The bitwise OR operator `|` is the result of the corresponding bits being combined with OR. To build off the previous example, x `|` y = 0001 because 0 `|` 0 = 0 and 0 `|` 1 = 1.

- Bitwise XOR (^) is a little more interesting since it introduces exclusively. Either one of the bits or the other, but not both, can be true for XOR to return true. In the above example, x ^ y = 0001.
- The bitwise NOT operator (~) simply flips all the bits. ~x = 1111.
- The left shift operator (<<) shifts the bits left by the specified number. y << 1 = 0010. Generally, the extra bits are padded with zeroes. Interestingly, the left shift operator is equivalent to the mathematical operation of multiplying by 2.

5.2 Real-world Applications

- If you've ever lost data because your hard drive malfunctioned, you might be interested to learn about RAID arrays, which backup data by storing it redundantly on two disks. In a RAID 1 setup, for example, two separate hard drives are treated as identical by the operating system so that any data written to one is also written to the other. In the case that one fails, then, all of the data is intact on the other.
- RAID 5 goes one step further. Given three disks of identical storage space, say 1 TB, a RAID 5 configuration will give the illusion of having a single 2 TB disk. Two of the three disks are used to store the actual data and the third is used to store a checksum, which is actually nothing more complicated than the XOR of the bits on the two data disks. Conceptually, the XOR operation can be applied in reverse. If some bits on one of the data disks are corrupted or lost, they can be restored by applying XOR to the corresponding bits on the two undamaged disks—the one undamaged data disk and the checksum disk.
- Another real-world application of bitwise operators, specifically XOR, is the ability to swap the values of two variables without using a temporary variable. Take a look at `swap2.c`:

```
/**************************************************************************
 * swap2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Swaps two variables' values.
 *
 * Demonstrates (clever use of) bitwise operators.
 **************************************************************************/

#include <stdio.h>
```

```
// function prototype
void swap(int *, int *);


int
main(int argc, char *argv[])
{
    int x = 1;
    int y = 2;

    printf("x is %d\n", x);
    printf("y is %d\n", y);
    printf("Swapping...\n");
    swap(&x, &y);
    printf("Swapped!\n");
    printf("x is %d\n", x);
    printf("y is %d\n", y);
}


/*
 * void
 * swap(int *a, int *b)
 *
 * Swap arguments' values.
 */

void
swap(int *a, int *b)
{
    *a = *a ^ *b;
    *b = *a ^ *b;
    *a = *a ^ *b;
}
```

We're passing `x` and `y` by reference, you'll notice. Then the magic happens, which we'll leave you to figure out at home. Of course, you can just follow an example line by line in the lecture slides. Seriously, do I have to do everything for you?[3]

- One last real-world application of bitwise operators that we'll discuss is bit masking. To demonstrate this, let's try writing a program called `binary` which takes an `int` from the user and displays it in binary notation:

[3] No, seriously, do it yourself.

```
#include <cs50.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    // prompt user for number
    int n;
    do
    {
        printf("Non-negative integer please: ");
        n = GetInt();
    }
    while (n < 0);

    // print number in binary
    for (int i = sizeof(int) * 8 - 1; i >= 0; i--)
    {
        int mask = 1 << i;
        if (n & mask)
            printf("1");
        else
            printf("0");
    }
    printf("\n");

    // that's all folks
    return 0;
}
```

First, we're executing a do-while loop to prompt and reprompt the user until he provides a non-negative integer. Conceptually, we know an `int` is represented under the hood as 32 bits. So we need a loop that iterates 32 times. Hence initializing `i` to `sizeof(int) * 8 - 1`, or 31, and decrementing to 0—this also has the effect of iterating from left to right (least-significant to most-significant).

- On every iteration, we want to ask if the current bit is 1 or 0 and print it accordingly. To do this, we'll use a bit mask which is 32 bits in memory filled with 0's at every location except our current location, which is filled with a 1; we generate this mask by shifting the bit 1 left by a number of places equal to 31 minus the number of iterations of the loop that have executed so far (`mask = 1 << i;`).
- By combining this bit mask against the user-provided `int` using the `&` operator, we'll get true returned whenever there is a 1 in the current

location of the user-provided `int`. When we do get true, we'll print out the ASCII character 1.

- So if the user provides the number 15, then on the first iteration of the loop, we'll have:

```
n =    00000000000000000000000000001111
mask = 10000000000000000000000000000000
```

  Thus, `n & mask` will return 0. If `n & mask` returns anything non-zero (i.e. true), it's because there's a 1 in the current bit place of `n`.

- In Problem Set 4, you most likely used a 2-D array of `bool`'s to store the original Sudoku board. If each of these `bool`'s was implemented as a `char`, you were using 7 more bits than were necessary at each of the 81 locations in the array.

- We've seen that bitwise AND serves as a getter for bit values. Bitwise OR can actually serve as a setter for bit values. For example, `x = x | 1` will set the value of x to be 1.

6 Hash Tables (67:00–73:00)

- For Problem Set 6, we'll be handing you a dictionary of a hundred thousand words which you will need to load up into memory in order to spellcheck several written works. But what data structure will you use to store the dictionary in memory? Certainly a linked list won't be adequate because your running time for insertion or lookup will be linear.

- One data structure that might prove useful to you for Problem Set 6 is a hash table. Much like an array, it provides instant access to its elements. Where you might store each word in the dictionary will be determined by a hash function. A very naive hash function, for example, might assign all words starting with the letter 'a' to the 0th location in the hash table. But then you run into the problem of *collisions*. What will you do in the case that there are multiple words that begin with 'a'?

- Using the approach of linear probing to handle collisions, you would attempt to place the second word beginning with 'a' in the 0th location of the hash table and, upon finding that location filled, you would walk through the table one location at a time until you found an empty location in which to put it.

- The problem with this linear probing approach, of course, is that lookup will run in $O(n)$, the worst case being that all the words begin with the same letter and you have to step through the hash table one element at a time until you determine whether the lookup word is stored or not.

- The motivation for hash tables is to provide constant-time lookups. With a well-designed hash function, there will be minimal collisions and after a simple mathematical operation, you will know exactly where in the hash table to look in order to find a word.