Computer Science 50
Fall 2010
Scribe Notes

Week 7 Monday: October 18, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 7 Monday: October 18, 2010
Andrew Sellergren

1 Announcements and Demos (0:00–2:00, 15:00–16:00, 20:00–21:00)

- This is CS50.
- 1 new handout.
- This week we'll continue our work in C, but next week we'll begin working with HTML and CSS, two mark-up languages, as well as PHP and JavaScript, two programming languages, as we tackle web development.
- The Harvard Science Review is looking for a webmaster. Contact them if you're interested.
- On Wednesday, we'll release a new and improved version of Harvard-Courses that fixes some of the more griped-about bugs and implements some of the more asked-for features.

2 Problem Set 5 (2:00–15:00)

- You may already have seen the image distributed with Problem Set 5 that looks like red noise. This is an example of steganography, the art of hiding messages in pictures and graphics. Within this image is a line of cyan-colored text which you'll be asked to reveal.
- Last lecture, we introduced the ability to store data persistently on disk. For this problem set, you'll be manipulating a chunk of data and writing out JPEG files that you recovered from it. A very simple black-and-white image could be stored on disk as a series of bits, ones being white and zeroes being black. Each bit would represent a single pixel, that is, one very small component of the image. With more complicated images, metadata such as the height and width of the image are stored in the header of the file. For a bitmap file, a struct called `BITMAPFILEHEADER` stores this metadata. The rest of the file is laid out in a standardized way such that operating systems will know how to read the bytes that comprise it.
- In addition to `BITMAPFILEHEADER`, we'll be defining a few other useful data types in order to work with bitmaps. We'll define a `WORD`, for example, to be a `uint16_c`. The "u" stands for "unsigned," meaning the integer only stores positive numbers and doesn't use a bit to represent its signage. The 16 indicates that the size of this integer doesn't vary between systems, but rather is always 16 bits. Likewise, a `BYTE` will designate a `uint8_c` which will always store positive numbers. We don't use `char`'s here because we don't want our bits to be converted to numbers and letters in this case.
- After David took a series of pictures of staff members, he "accidentally" deleted them from his flash card. However, as we've learned, deleting a file doesn't necessarily erase the contents of the file, but only tells the

operating system to forget where the file is stored. Thus, once we get a forensic image of the flash card, that is, a complete dump of the bits from the card, we'll be able to scan over it looking for chunks of data that are stored in the JPEG file format. Whenever we find such a chunk of data, we'll write it out to a file. In the end, we'll have recovered 50 JPEGs, all of which were previously "deleted" from the flash card.

- The second, optional challenge for this problem set is to find the subjects of the images from the flash card and photograph yourself with them. The section that does this first with all of the subjects will win an amazing prize!
- In this problem set, you'll also be tasked with writing a program to resize an image. In the standard edition, you'll be asked to enlarge the image, but in the Hacker Edition, you'll be asked to shrink it down. If you take on the Hacker Edition, you'll have to make some intelligent decisions about which pixels to throw away.
- In order to examine the output of your program and verify that the images it recovered are valid and correct, you'll need to use Secure File Transfer Protocol (SFTP) which is similar to SSH but allows you to transfer files to and from a remote computer. On a Mac, you can use a program called Cyberduck and on a PC, you can use a program called WinSCP to connect to the CS50 Cloud via SFTP. More detailed instructions are available in the problem set specification and on the Resources page.

3 Common Mistakes from Quiz 0 (16:00–20:00)

- Realize that if you declare a string, it is wrong to dereference it when you pass it to `printf`.

  WRONG:

  ```
  char *s = "hello";
  printf("%s\n", *s);
  ```

  RIGHT:

  ```
  char *s = "hello";
  printf("%s\n", s);
  ```

  `*s` is actually retrieving the first character of `s`, so this syntax is valid (although it doesn't achieve what we want in this case) if you use the format string `%c`.

- Technically speaking, `NULL` is a pointer whereas `'\0'` is a `char`.

  WRONG:

```
char *s = "hello";
while (s[i] != NULL) {
   i++;
}
```

RIGHT:

```
char *s = "hello";
while (s[i] != '\0') {
   i++;
}
```

- Two other important distinctions which are detailed in the quiz solutions are those between `\n` and `\r` and between `#include` and `-l`.

4 From Last Time (21:00–33:00)

- Linked lists proved to be advantageous over arrays because insertion into linked lists is in $O(n)$ and linked lists do not have a fixed size.
- However, linked lists do not allow for random access (i.e. jumping to a specific element) as arrays do.
- Another type of data structure we've looked at is a stack, which demonstrates LIFO storage, i.e. the first element to be removed is the last one that was inserted. Alternatively, a queue demonstrates FIFO storage, i.e. the first element to be removed is the one that was inserted before all the others, like a line at an amusement park, for example.
- To implement a queue, you could actually just use an array. We'd then need to implement a function that retrieves the next member that should be removed, like so:

```
struct queue
{
    int head;
    int members[100];
    int size;
};

int pop(queue *q)
{
    int next = q->head;
    q->head = (q->head + 1) % 100;
    return q->members[head];
}
```

As you can see, we need another piece of data in our struct to store the index of the member that should be removed next. In our `pop` function, we must also increment that index so that the next time `pop` is called, it will return a different person from the queue. To keep from iterating off the end of the array, we'll use the modulus operator. This assumes that the members of the array will be continually refreshed. It also assumes that the entire queue will be filled, which isn't necessarily true. To account for this, we need the `size` variable in our struct so that we can keep track of how full our queue is.

5 Valgrind (33:00–43:00)

5.1 `memory.c`

- Valgrind is a program which will help you track down memory-related bugs in your code. It will be especially germane to Problem Set 6, but let's take a look at an example now:

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


int
main(void)
```

```
{
    f();
    return 0;
}
```

This program demonstrates two memory-related bugs: first, touching memory past the bounds of an array and second, not freeing memory that was `malloc`'ed. Valgrind will reveal both of these bugs to us:

```
valgrind -v --leak-check=full a.out

==11201== Invalid write of size 4
==11201==    at 0x804840F: f (memory.c:23)
==11201==    by 0x8048421: main (memory.c:30)
==11201== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==11201==    at 0x4025BDC: malloc (vg_replace_malloc.c:195)
==11201==    by 0x8048405: f (memory.c:22)
==11201==    by 0x8048421: main (memory.c:30)
```

Valgrind gives us a large amount of output when we run it, but the part that we care about is excerpted above. This excerpt tells us that at line 23, we wrote 4 bytes (i.e. an `int`) improperly. This accords with our previous analysis of `memory.c` which indicated that we were touching memory past the bounds of the array `x`. The excerpt also tells us that 40 bytes of memory are "lost." This alludes to the memory leak we mentioned earlier since we failed to free the memory that we `malloc`'ed for `x`, whose size is 40 bytes since it has space for 10 4-byte integers. If we add the line `free(x)` to our program and rerun Valgrind, we'll see this error disappear.

- In Valgrind's output, we also see memory addresses written with the `0x` prefix. Recall that these are hexadecimal numbers. Why hexadecimal? Because 16 is a power of 2, we can conveniently represent two bytes with only two hexadecimal digits. For example, `0xff` represents the number 255, written as `11111111` in binary.

6 Bitwise Operators (43:00–63:00)

6.1 `binary.c`

- In order to manipulate the bits of an image so as to conceal a message in it, we can make use of bitwise operators. We've already looked at binary operators `&&` and `||` in the context of if conditions. It turns out that `&` and `|`, their bitwise counterparts, have very different meanings, as you may have discovered if you accidentally used them in place of `&&` and `||` in an if condition.

- `binary.c` introduces us to the use of bitwise operators:

```
/**************************************************************************
 * binary.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Displays a number in binary.
 *
 * Demonstrates bitwise operators.
 **************************************************************************/

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


int
main(void)
{
    // 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");
}
```

If we compile and run `binary`, we're prompted for a non-negative integer. Entering 3 gives us the following output:

```
00000000000000000000000000000011
```

As you probably guessed, this is the binary representation of the decimal number 3. Because a integer consists of 32 bits, we print out a 0 or a 1 for every bit, including leading zeroes.

- To do this conversion, we'll need to iterate from left to right over all of the bits in the integer. We begin by initializing `i` to 31, albeit in a roundabout way using `sizeof` since we want our code to be portable from system to system in case integers are implemented with 64 bits on a different architecture.
- On this system, the leftmost bit represents the largest number and thus is the highest-order, or most-significant bit whereas the rightmost bit represents the is the lowest-order, or least-significant bit.[1]
- Our variable `mask` will be initialized using `<<`, the left shift operator, which shifts all of the bits of a number to the left. This is easier to visualize with examples: if `i` is 1 (or 0001 in binary, with 4 places) and `mask` is initialized to 1, then `mask << i` gives us 2 (or 0010). If `i` is 2, and `mask` is initialized to 1, then `mask << i` gives us 4 (or 0100). If `i` is 3, then `mask << i` gives us 8 (or 1000). Here, in the first iteration of our loop, `i` is 31, so the 1 will be shifted 31 bits to the left.
- Once we've initialized `mask`, we're going to check the condition `n & mask`. The bitwise operator `&` will return a 1 for each bit that is set to 1 in both `n` and `mask`. Because `mask` has only a single bit set to 1, `n & mask` will return true only if `n` is set to 1 at the exact same bit. Thus, this condition is checking, in the first iteration of the loop, if the 31st bit of our user-provided integer is a 1. In the second iteration of the loop, it's checking if the 30th bit of our integer is a 1. And so on.

6.2 `tolower.c`

- One real-world application of bitwise operators is in converting letters from lowercase to uppercase and vice versa. In ASCII, the letter "A" is 65 (01000001) and the letter "a" is 97 (01100001). Notice that these numbers differ only by a single bit—the bit in the 32's place. We can leverage this like so:

```
/**************************************************************************
 * tolower.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Converts an uppercase character to lowercase.
 *
 * Demonstrates bitwise operators.
 **************************************************************************/
```

[1] This isn't true on all systems. It's a trait called endianness which we'll describe in more detail later.

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


int
main(void)
{
    // prompt user for an uppercase character
    char c;
    do
    {
        printf("Uppercase character please: ");
        c = GetChar();
    }
    while (c < 'A' || c > 'Z');

    // print number in lowercase
    printf("%c\n", c | 0x20);

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

The number `0x20` is 32 in hexadecimal. When we write `c | 0x20`, we're flipping the bit in the 32's place in `c`, which, as we just saw, is set to 1 in lowercase letters. So, to capitalize a letter, we simply change the bit in the 32's place from 0 to 1, which is equivalent to adding 32.

6.3 `toupper.c`

- Converting to uppercase requires the `&` operator as we in `toupper.c`:

```
/**************************************************************************
 * toupper.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Converts a lowercase character to uppercase.
 *
 * Demonstrates bitwise operators.
 **************************************************************************/

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

```
#include <stdio.h>


int
main(void)
{
    // prompt user for a lowercase character
    char c;
    do
    {
        printf("Lowercase character please: ");
        c = GetChar();
    }
    while (c < 'a' || c > 'z');

    // print number in lowercase
    printf("%c\n", c & 0xdf);

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

The `|` operator has the effect of flipping a bit from 0 to 1. If we want to flip a bit from 1 to 0, we can use `&` with 0 as one of the operands. However, we need to make sure to keep all the other bits in the number the same. We do this by setting all of the bits **except** the bit in the 32's place to 1. Take a look at the following example that uses "a", written as `01100001`:

```
   01100001
&  11011111
===========
   01000001
```

Each bit of the original letter, when passed to `&` with a 1 as the other operand, stays the same. The only bit that changes is the bit in the 32's place, which always flips to 0: anything passed to `&` with 0 as the other operand becomes 0. `11011111` in hexadecimal is `0xdf`.

7 A Teaser (63:00–66:00)

- Problem Set 6 will challenge you to implement a spellchecker using a dictionary of some 150,000 words. One approach might be to put every dictionary entry into an array and then use binary search to look up words. However, binary search is in $O(\log n)$, which isn't actually that fast, especially compared to $O(1)$. $O(1)$ is called constant time because it implies that only a single step is required to find the answer. A data

structure called a hash table, if you choose to implement it, will actually enable you to achieve constant-time lookup in your dictionary.

- Also of note: there will be an opt-in competition to see who can implement the fastest spellchecker in C. May the best computer scientist win!