Computer Science 50
Fall 2010
Scribe Notes

Week 5 Wednesday: October 6, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 5 Wednesday: October 6, 2010
Andrew Sellergren

1 Announcements and Demos (0:00–3:00)

- This is CS50.
- 0 new handouts.
- No lecture on Monday because of the holiday. The quiz will be on Wednesday during normal lecture time. Expect an e-mail regarding where you are to report based on your last name. If you don't receive this e-mail for some reason, check the course website.
- The music you heard on your way into Sanders was by way of special request. Joanna wanted to celebrate her two-year anniversary with her boyfriend David by playing one of his favorite DDR songs. Congrats! The song, in fact, commemorates the Konami Code which unlocked a cheat in the original Contra game for NES. Try entering the Konami Code on the course website![1] For those of you who don't know what Contra is, check out this video.
- Speaking of games, check out this XKCD cartoon which you should now understand.

2 The CS50 Library (3:00–21:00)

- Although it's not important that you understand every line of code in the CS50 Library at this point in the semester, you should have a general sense of what it does.
- The CS50 Library consists of two files: `cs50.h` and `cs50.c`. `cs50.h` is the header file that contains the function prototypes and a slew of comments. `cs50.c` is the file that contains the actual function definitions.
- At the top of `cs50.h`, we use a trick with `#ifndef` and `#define` to check if a constant called `_CS50_H` has already been defined. If it hasn't, then we define it as the entirety of the header file. We do this to prevent the header file from being included more than once.
- Within `cs50.h`, we include a few other header files, among them `stdbool.h`, which defines the constants `true` and `false`.
- The line `typedef char *string` makes the keyword `string` an alias for the variable type `char *`.
- As we saw last time, `GetString` is defined such that we grow a buffer as the user provides more and more input. We do so by calling `malloc` and `realloc` to ask the operating system for more memory.

[1] And in Google Reader!

- At the end of `GetString`, we do some cleanup whereby we ask for another buffer that's exactly the size of the user's input (plus an additional byte for the null terminator), copy the user's input into it, and then free the original buffer.

- The other functions in `cs50.c` depend on `GetString` to obtain the user's input. We then pass the user's input, stored in `line`, to a function `sscanf` which looks for format strings within it. In this case, we're looking for an integer followed by a character. What we're checking, however, is if only the integer was found, in which case `sscanf` will return 1, the if condition will evaluate to true, and the integer will be returned. If both an integer and a character are captured, `sscanf` will return 2 and the user will be asked to retry.

- We actually have to pass `&n` and `&c` because if we pass `n` and `c`, the values extracted from the user's input will be lost when `sscanf` returns, just as we saw with `buggy3.c`.

- Question: what will `sscanf` return if the user inputs only characters? `sscanf` will return 0 in this case because it will not have been able to populate `%d` first.

2.1 `scanf1.c`

- `scanf1.c` demonstrates the traditional way of obtaining user input via `scanf`:

```
/*****************************************************************************
 * scanf1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Reads a number from the user into an int.
 *
 * Demonstrates scanf and address-of operator.
 ****************************************************************************/

#include <stdio.h>


int
main(void)
{
    int x;
    printf("Number please: ");
    scanf("%d", &x);
    printf("Thanks for the %d!\n", x);
}
```

When we compile and run `scanf1`, we see that it works as long as we provide integer input. However, if we enter something like "abc," the program tries to cast our input to an integer and ultimately demonstrates unexpected behavior. This is why we introduced the CS50 Library: to save you the trouble of implementing these functions yourself.

2.2 `scanf2.c`

- Try to spot the bug in `scanf2.c`:

```
/****************************************************************************
 * scanf2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Reads a string from the user into memory it shouldn't.
 *
 * Demonstrates possible attack!
 ***************************************************************************/

#include <stdio.h>


int
main(void)
{
    char *buffer;
    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the \"%s\"!\n", buffer);
}
```

  The variable `buffer` is an uninitialized pointer. Think back to Binky: we have a pointer, but not a valid pointee, which is bad news. When we try to interpret the junk that's in `buffer` as a memory address, the program fails with a segmentation fault. Even worse, we might be able to trigger a buffer overrun attack.

- Question: is the value in `buffer` being interpreted as a memory address? Yes, and to prove it, we can print it out using `printf` with the format string `%d`. When we do so, we get a number like 2719732.

- One possible defense against this would be to allocate `buffer` as an array with a large number of bytes. Even then, however, if the user knows what that number of bytes is, he can exploit your program by entering input longer than that. The real solution is to code defensively as we did in the CS50 Library.

3 Linked Lists (21:00–50:00)

- When we faced the problem of storing many similar variables, we discovered the convenience of an array. Not only does an array prevent us from having to declare variable after variable of the same type, it allows us random access to its elements using bracket notation. That is, we can immediately find an element in the array if we know its index.

- One thing that arrays are not good for is insertion. We saw this during our sorting demos when we wanted to move an element from the beginning to the middle. In order to do so, we had to swap it with another element in order to avoid having to shift all the in-between elements down by one.

- `GetString` highlights another shortcoming of arrays: they are of finite size. You have to know in advance how big they need to be. If you guess wrong, you must go through the expensive operation of allocating more memory and copying the old memory into the new. Generally, anytime you need to contact the operating system, as you do when you request more memory, you incur a performance penalty.

- Enter linked lists. Each of the nodes in a linked list consists of both a value and a pointer to the next nodet in the list. This data structure is compelling because insertion is very fast. Instead of a large chunk of contiguous memory, we now have scattered small chunks of memory that are stitched together by pointers.

- In order to define a linked list node, we'll turn to the syntax for structs that we introduced last time:

```
typedef struct node
{
    int n;
    struct node *next;
}
node;
```

  Notice that the syntax is slightly different from what we used to declare a `student` struct. By using the above, we can declare a new `node` instead of a new `struct node`, which are actually the same thing, the former simply being shorter. The pointer in this struct is going to be pointing to another one of itself, that is, another `node`.

3.1 File I/O

- Last time, we used a function `fprintf` to write data to an output file. This was our first foray into file I/O, or file input/output. For Problem Set 5, you'll be manipulating files in order to recover a series of JPEGs from a formatted flash drive. For simplicity's sake, these JPEGs will be contiguous chunks of memory. However, on most operating systems, files are not stored as large contiguous chunks, but rather as small scattered chunks. For that reason, files are implemented as linked lists.

- Because files are implemented as linked lists, it's very easy for them to become fragmented and for those fragments to become "lost" on disk. If you delete a pointer that's pointing to a node of a linked list, then you have no way of accessing the memory that was once stored in that node. Whatever was stored in that memory is not actually deleted, however, but only orphaned. When for performance reasons, the operating system gives you a larger chunk of memory than you need, say 512 bytes for 200 characters, the memory you don't use—the slack space—often contains remnants of these orphaned memory chunks. These are honeypots for forensic data analysts who are investigating a hard drive as evidence.

- To elaborate, when you delete a file on your computer, the actual bits that comprise the contents of the file remain intact. All that is deleted is the entry in the directory table, which is simply a list of file pointers maintained by your operating system. Thus, you can recover the original contents of the file if you can scan the hard drive and you know what you're looking for, as we will do in Problem Set 5.

- Mac OS is better than other operating systems in that it offers an option to "Secure Empty Trash," which erases the entry in the directory table as well as zeroes out the actual memory corresponding to the file.

3.2 Operations

3.2.1 Search

- In order to visualize a linked list, we'll ask 5 volunteers to come on stage and hold pieces of paper with numbers on them. In addition to holding the pieces of paper, the volunteers will point with their left hand to another volunteer to signify the `next` pointer pointing to another node in the list. The last volunteer will point to no one since he is at the end of the list. The values in the list will be in sorted order from minimum to maximum. In order to store a linked list, we need only to store a single pointer to the first node of the list; let's call it `first`. From `first`, we can find all the other nodes in the list simply by following their `next` pointers.

- While optimal for insertion, linked lists are suboptimal for access. Arrays offer random access, but linked lists require stepping through the elements one at a time. In order to do so, we must use arrow notation to both access and dereference the `next` pointer inside each node.

- Searching for a value in the linked list is a matter of traversing the linked list and checking each value along the way.

3.2.2 Insertion

- To begin the process of inserting a value into the linked list, we `malloc` enough memory to hold a new node and place in it the value of interest. We'll call the pointer to this memory (as returned by `malloc`) `newptr`.

- As with search, we traverse the list, this time checking each value to see if it is larger than the value we want to insert. If we reach the end of the list and find that no values are larger than the value we wish to insert, then we assign `newptr` to the current last node's `next` pointer (thereby pointing it to the chunk of memory we just `malloc`'ed) and we assign `NULL` to the `next` pointer of our newly inserted last node.

- If we need to insert a node at the beginning of the list, we take only two steps (after traversing to determine that the number belongs at the beginning of the list). First, we assign `first` to the `next` pointer of our newly allocated node. Second, we point `first` to our newly inserted node. The order of these steps is important. If we first pointed `first` to our newly allocated node, we would lose track of our entire linked list!

- Insertion into the middle of a linked list is the most complicated case. Again, we traverse the list until we find the right position for our new node. We'll see in this case that we need two pointers when we traverse the list, the first to point to the current node (`ptr`) and the second to point to the previous or predecessor node (`predptr`). When the current node's value is larger than the value of the new node we're inserting, we know that the new node belongs after `predptr`. In order, we point the new node's `next` pointer to `ptr` and `predptr`'s node's `next` pointer to the new node. Whew.

- Because we have to traverse the list each time we want to insert an element, insertion to a linked list is in $O(n)$. That is, in the worst case, we'll have to take n steps through the list in order to insert a node at the end.

3.2.3 Deletion

- To delete a node in the middle of the list, we point a temporary node such as `ptr` to the node we wish to delete. We then point `predptr` to the node after the node we wish to delete. Finally, we call `free` on the node that `curr` is pointing to.

- To delete a node at the end of the list, we point `ptr` to the last node in the list and update the `next` pointer of `predptr` to point to `NULL`. We then call `free` on `ptr`.
- To delete a node at the beginning of the list, we begin by pointing `predptr` and `ptr` at the first and second nodes in the list, respectively. We then call `free` on `predptr` and update `first` to point to `ptr`.
- In all these cases, the order in which we free nodes and update pointers is extremely important so that we don't orphan single nodes or entire segments of the list.

3.3 `list1.c`

- Let's see how we might implement linked list operations in actual C code. The various operations which are associated with linked lists are displayed on the start-up menu of `list1`:

  ```
  MENU

  1 - delete
  2 - find
  3 - insert
  4 - traverse
  0 - quit
  ```

  The program itself enters a `do while` loop that waits for the user's menu selection. Once the user has made a selection, `list1` calls the appropriate function using a `switch` statement:

  ```
  // get command
  printf("Command: ");
  c = GetInt();

  // try to execute command
  switch (c)
  {
      case 1: delete(); break;
      case 2: find(); break;
      case 3: insert(); break;
      case 4: traverse(); break;
  }
  ```

  Let's assume we chose 3 for "insert":

  ```
  /*
   * Tries to insert a number into list.
  ```

```
 */

void
insert(void)
{
    // try to instantiate node for number
    node *newptr = malloc(sizeof(node));
    if (newptr == NULL)
        return;

    // initialize node
    printf("Number to insert: ");
    newptr->n = GetInt();
    newptr->next = NULL;

    // check for empty list
    if (first == NULL)
        first = newptr;

    // else check if number belongs at list’s head
    else if (newptr->n < first->n)
    {
        newptr->next = first;
        first = newptr;
    }

    // else try to insert number in middle or tail
    else
    {
        node *predptr = first;
        while (true)
        {
            // avoid duplicates
            if (predptr->n == newptr->n)
            {
                free(newptr);
                break;
            }

            // check for insertion at tail
            else if (predptr->next == NULL)
            {
                predptr->next = newptr;
                break;
            }
```

```
                // check for insertion in middle
                else if (predptr->next->n > newptr->n)
                {
                    newptr->next = predptr->next;
                    predptr->next = newptr;
                    break;
                }

                // update pointer
                predptr = predptr->next;
            }
        }

        // traverse list
        traverse();
    }
```

- Once `malloc` returns, we do a sanity check to make sure it didn't return `NULL`. If it did and you were to try to dereference it, your program would seg fault. This is a feature of C.[2] Essentially, the compiler is preventing you from accessing the memory address 0x0, even though it does exist.
- In the next few steps, we follow those which we walked through a few moments ago. We take in the user input and assign it to the new `node` in addition to pointing its `next` pointer to `NULL`. We do so using the arrow notation that we mentioned briefly:

```
  // initialize node
  printf("Number to insert: ");
  newptr->n = GetInt();
  newptr->next = NULL;
```

- First, we check if the list is empty, which is the easiest case. If the list is empty, the node we're inserting becomes the entire list.
- Next, we check if the new node belongs at the head of the list, i.e. if its value is less than the value of the first node:

```
  // else check if number belongs at list's head
  else if (newptr->n < first->n)
  {
      newptr->next = first;
      first = newptr;
  }
```

[2] When you become an enterprise-level programmer, everything's a feature and nothing's a bug.

`first` is a pointer to a node, so we must use the arrow notation to access the value it stores in `n` and compare it to the value of the node we're inserting. If `first`'s value is greater than the value to be inserted, we first point the `next` pointer of `newptr` at `first` and then point `first` to `newptr`. If we reversed the order of these two steps, we would orphan the entire list.

- When we tackle the middle and end cases, we'll need to bring in `predptr`, which plays the same role as we did when we traversed the list looking for the place to insert the new node. First we're checking for duplicates and if we find the value is already inserted, then we won't insert it again.
- The end case, or tail case, is easier to handle than the middle case. All we need to do is point the `next` pointer of `predptr`, which is pointing at the last element of the list, to the new node.
- The middle case is perhaps the most complicated, yet it only actually requires two pointer updates. Thus, we've only seen two general cases: one that requires a single pointer update and one that requires two pointer updates. We'll gloss over the middle case for now, but do dive into the code we've provided and reconsider the examples we walked through with our volunteers.
- `find()` is a little less complicated than `insert()`:

```
/*
 * Tries to find a number in list.
 */

void
find(void)
{
    // prompt user for number
    printf("Number to find: ");
    int n = GetInt();

    // get list's first node
    node *ptr = first;

    // try to find number
    while (ptr != NULL)
    {
        if (ptr->n == n)
        {
            printf("\nFound %d!\n", n);
            sleep(1);
            break;
        }
        ptr = ptr->next;
    }
}
```

Here, we keep walking through the list as long as `ptr` isn't `NULL`, that is, as long as we're not at the end of the list. During each iteration of the loop, we compare the current node's value against the user-provided input. If the two are equal, then we announce as much and break out of our loop.

- Because this is a `while` loop, we must take care of the iteration explicitly:

  ```
  ptr = ptr->next;
  ```

  This is much like `i++` in the other `while` loops we've looked at. We're taking `ptr`, which points at the current node, and reassigning it to point to the next node.

- Other data structures include stacks, which exhibit last-in-first-out (LIFO) storage, and queues, which exhibit first-in-first-out (FIFO) storage.