Announcements (0:00-2:00)

- Do yourself a favor and start problem sets early. (They'll be a lot more fun!)
- We'll be raffling off a copy of Office Ultimate 2007 on Wednesday, November 01
- Quiz 1 will be given on Wednesday, November 07

Growing a Buffer (2:00-15:00)

- Recall that GetInt(), GetString(), etc., are not built into C, but rather are written in the CS 50 library for your convenience
- Let's take a look at the implementation of GetString().
- So far, the function we've seen to get input from the user is scanf()
- Problem with scanf() is that you have to pass it a buffer into which it should put whatever it reads in. That buffer must necessarily be of finite size. What if the user enters something greater than the buffer size? We could end up with buffer overflow and a segmentation fault.
- To avoid this problem our strategy in GetString() is to allocate some space for the buffer initially, but to *grow* the buffer if that ends up not being enough space.
- Here's the code that allocates space initially:

```
size = SIZE;
buffer = (string) malloc(size * sizeof(char));
if (buffer == NULL)
    return NULL;
```

- SIZE is #defined in cs50.h to be 128. We make a reasonable assumption that the string entered by the user is likely to be less than 128 characters long.
- We allocate 128 chars and cast that memory into a string to be explicit about the kind of pointer buffer will be. Remember that string is a char *, or pointer to a character.
- Then we check to make sure that buffer is not NULL. That could happen if we were asking for too much memory.
- Now we start reading in characters:

```
n = 0;
while ((c = fgetc(stdin)) != '\n' && c != EOF)
{
    /* grow buffer if necessary */
    /* this part omitted */

    buffer[n++] = c;
}
```

- Let's look at the while condition to start off with. fgetc() is "file get character". We're getting input from stdin, a built in file that refers to the user's keyboard. We keep getting characters until we encounter a newline or an EOF (end of file) character.

- Now, so long as n does not equal size-1, we put the next character into the next slot of the buffer. That's all that buffer[n++] = c means.
- But when n does get to size-1, the buffer is full. In that case we have to "grow" the buffer.
- Here's the code for that (omitted above):

```
if (n == size - 1)
{
    /* keep buffer size within unsigned int range */
    if (size > (UINT_MAX / 2))
    {
        free(buffer);
        return NULL;
    }

    /* double buffer's size */
    size *= 2;
    tmp = (string) malloc(size * sizeof(char));
    if (tmp == NULL)
    {
        free(buffer);
        return NULL;
    }
    strncpy(tmp, buffer, n);
    free(buffer);
    buffer = tmp;
}
```

- The catch is that we can't simply tell the operating system to extend the buffer to whatever memory happens to be next to it. There might be something already occupying that space.
- Instead, we have to ask for a bigger buffer somewhere else, and copy over everything we have into the beginning of that buffer.
- If the size is greater than half the largest possible int (UINT_MAX/2), then we have to bail.
- Otherwise, double the size and malloc the new size. Assign the newly allocated memory to tmp, make sure tmp's not NULL, and copy over what we have into tmp.
- strncopy() copies the contents of buffer into tmp but only up to n characters
- This is better than strcopy() because it checks the bounds
- Finally, we reassign our new, bigger array tmp into the variable buffer
- So, why did we choose 128 as our starting size?
- We could have started with 1 or 2 and just doubled it as needed. But this would result in a lot of work if we had large input. malloc() and free() are very expensive operations that can hurt the performance of your program if you call them too much.
- 128 seems kind of arbitrary, but represents a carefully made design decision. The idea is that growing the buffer is a pretty costly operation that wants to be avoided if at all possible.

Singly-Linked Lists (15:00-27:00)

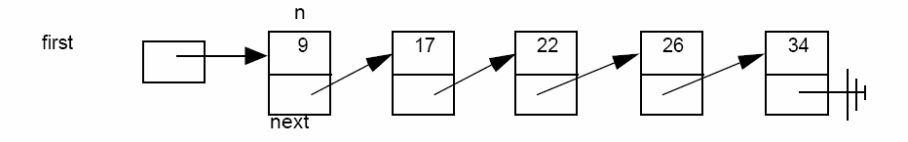
- Let's talk about arrays. What are some advantages and disadvantages of storing data in an array?
- Advantage: random access. You don't have to iterate up to the value you want. You can just jump to it using bracket notations. We call this gives O(1) or constant time lookup. You can do binary searches and other fast algorithms because of random access
- Advantage: simplicity
- Disadvantage: you need to know in advance how much space you need, unless you're willing to grow it's size later on
- Disadvantage: delete operations are costly. You have to shift everything down to cover up the vacated slot and then ignore the rightmost element or copy over to a smaller array.
- Disadvantage: insert operations similarly costly. You need to grow the array the way we did in GetString()
- So there's good things and bad things about arrays.
- Conceptually, how can we address the insert and delete problems? We want to have a data structure where the memory is not necessarily contiguous, but each element "leads" or "points" you to the next.
- We imagine our data structure as a row of boxes, just like an array. This time, however, the boxes are not necessarily continuous, they might be all over the place in memory.
- Each box stores two pieces of information: an int and a pointer to another box. This is how they all become linked together without being next to each other in memory.
- When we get rid of the assumption of contiguous memory, insertion and deletion become much easier
- We call such a data structure a singly-linked list
- We refer to the list by a pointer (usually called "first" for simplicity) that points to the first element of the list
- Remember when we defined a student structure last week using struct syntax?

```
typedef struct
{
    int id;
    char * name;
    char * house;
}
student;
```

- We deploy the singly-linked list using the same syntax:

```
typedef struct _node
{
    student * student;
    struct _node *next;
}
node;
```

- The only difference here is that we are sort of giving the structure two names: _node and then node.
- The reason for this is that we wish to make one of the data fields a pointer to another node. This results in a circular definition: we must refer to a node in the definition of the node!
- We can't use the word node inside the definition because it's not completely defined until the curly braces close. Therefore, we give it a kind of termporary name _node before the curly braces and use this name to refer to it within the definition.
- After the curly braces close, the structure is now defined as node and we never use the name _node again.
- The convention of using the same name preceded by _ is not necessary, but is a convention in C.
- We declare the pointer to our list as node * first = NULL;
- This is all you need to remember to keep the list around, just as with arrays, all you need to remember is the address of the first element or the name of it (say, a)



Manipulating Linked Lists (27:00-34:00)

- Let's make a program to manipulate linked lists
- Here's a start:

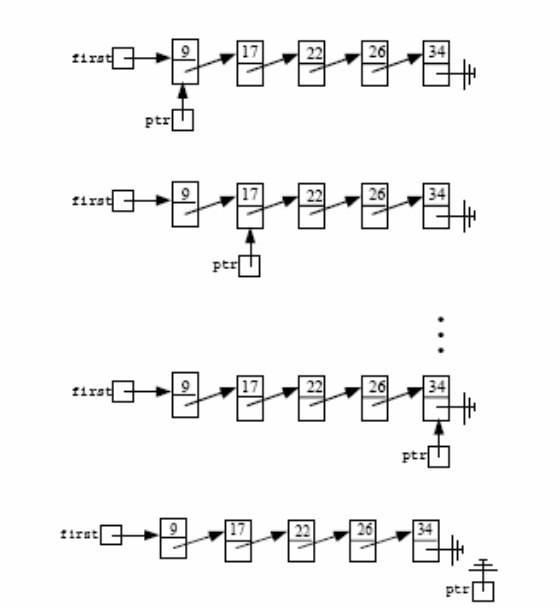
```
int
main(int argc, char * argv[])
{
    int c;
    do
    {
        // print instructions
        printf("\nMENU\n\n"
               "1 - delete\n"
               "2 - find\n"
               "3 - insert\n"
               "4 - traverse\n"
               "0 - quit\n\n");

        // 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;
            }
        }
        return 0;
}
```

- All this does is create a menu of options, display it to the user, read in the option that they choose, and then call the appropriate function
- We also declare our list at the top of the program (not shown): node * first;
- Now we have to write functions to accomplish the possible tasks the user might want to do

Traversing a Linked List (34:00-41:30)

- Alright, so first let's think about traverse(). What does it take to traverse the list?
  - Start at first.
  - While pointing at valid node, follow the pointer.
  - Print the value or the node you're at.
  - Now follow that node's pointer and repeat.
- Here's a graphical representation:



- Now let's think about the code.
- We'll need something to point with, a temporary pointer. Call it node * ptr.
- Then we'lll need a loop to iterate through the nodes. Start by pointing at first, and keep going until ptr is equal to NULL. While in the loop, just print out the value of the current node.
- ptr->n maens that we want the n datum from ptr . The reason we're not using * here is that *ptr refers to the whole chunk of memory. We're also not using ptr.n because ptr is a pointer,

whereas . syntax is for actual structures. So we could have also said (*ptr).n but -> is just concise and stylistically preferable.

- What's the update step? Go to the next field of the current pointer: ptr = ptr->next.
- Here's the code:

```
void
traverse()
{
    // traverse list
    printf("\nLIST IS NOW: ");
    node * ptr = first;
    while (ptr != NULL)
    {
        printf("%d ", ptr->n);
        ptr = ptr->next;
    }

    // flush standard output since we haven't outputted any newlines
yet
    fflush(stdout);

    // pause before continuing
    sleep(1);
    printf("\n\n");
}
```

Searching a Linked List (41:30-46:30)

- Now let's write find()
- Our goal now is to take a list and search for an element. If it's there, print YES. Otherwise, print NO.
- First, we ask the user what they are looking for and read that into a variable.
- Again, we'll start with a temporary pointer that we initialize to first: node * ptr = first;
- Then we're going to have a loop that continues so long as we're not at a NULL pointer.
- If n equals the value of that node, print "YES!" and break out of the loop.
- Otherwise, go to the next node.
- Here's the code:

```
void
find()
{
    // 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;
        }
    }
```

- Check out Wednesday's lecture for implementations of delete() and insert()!