CS 50: Introduction to Computer Science I
Harvard College
Fall 2007

Scribe Notes
Week 5 Wednesday
Anjuli Kannan

Working with Pointers (0:00-6:00)

- We can demonstrate the use of pointers with the following sample program:

```
int main (int argc, char **argv)
{
        int x;
        int * p;
        x=5;
        printf("%d\n",x);
}
```

- If we make and run, we see that this program just prints out 5.
- What if we want the address of x? Change the sixth line to `printf("%d\n", (int) &x)`. Make and run, and we get `bfcd2ccc`, a memory address
- Next, let's try setting p to point to x. What if again want the address of x? We can just print out the value of p:

```
int main (int argc, char **argv)
{
        int x;
        int * p;
        x=5;
        p = &x;
        printf("%d\n",(int) p);
}
```

- Again, we get a memory address, but this time it is a different address than we got before. Why? Because it's a different run of the program, so x has been put in a different location.
- What if we want to print the actual value of x without saying print x as before?

```
int main (int argc, char **argv)
{
        int x;
        int * p;
        x=5;
        p = &x;
        printf("%d\n",*p);
}
```

- This means we go to the address stored in p, and print the value there contained

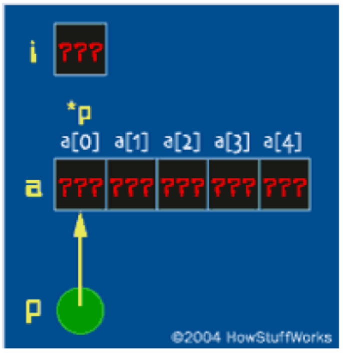
Arrays as Pointers (6:00-12:30)

- You are familiar with arrays from (a) representing your game board in 15 and (b) working with strings

- You can think of an array as a contiguous chunk of memory
- Suppose your array is called a. Then a is really just a pointer to that memory.
- Consider these lines of code

```
int i;
int a[5];
int *p = a;
```

- This results in the following:



- p is a pointer to the array; it contains the address of the array
- a is also a pointer to the array. After the third line of code is executed, a and p both contain the same address.
- Note that if we did this in two separate lines, we would declare `int * p;` and then say `p = a;`
- If we said `int * p` and then `*p=a;` we would be going to the address contained in p and putting in it the address contained in a (an integer). Note that we have no idea what address is initially stored in p, so this would likely result in a segmentation fault.

Strings as Arrays (12:30-26:30)

- Take a look at compare1.c:

```
int
main(int argc, char * argv[])
{
    // get line of text
    printf("Say something: ");
    char * s1 = GetString();

    // get another line of text
    printf("Say something: ");
    char * s2 = GetString();

    // try (and fail) to compare strings
    if (s1 == s2)
        printf("You typed the same thing!\n");
    else
        printf("You typed different things!\n");
}
```

- This program prompts the user for two strings and then determines whether the strings are the same or different
- This time, instead of storing the strings in variables of type “string”, which is defined only in the cs50 library, we say char * string1 = GetString(). This is because strings are actually just character arrays. We define a string type in cs50.h just to cover up some of the messy details of the language.
- GetString() makes an array of characters and returns a pointer to the first character
- Say our array of characters looks like this:

f	o	o	\0
0x4	0x8	0x12	0x16

- Then when we say char * s1 = GetString(), s1 gets the address of the first character in the string, 0x4
- Knowing the address of the first character is sufficient because (a) the characters are contiguous and (b) there’s always a null character to signal when we’ve gotten to the end.
- Now our program checks for equality by saying if(s1==s2). Is this code going to work?
- No, because each time the user types in the string, they get put in different places in memory
- s1 == s2 is perfectly valid syntax, but it just compares values of *pointers*, i.e. the addresses of the strings
- So how do we actually compare the two strings?
- Suggestion: *s1==*s2
- Well, this actually goes to the addresses and compares the values there, so it should work, but testing reveals that “foo” and “frank” are the same string. Why?
- *s1==*s2 is only comparing the first letter of each string because s1 and s2 are pointers to only the first characters
- What we need to do is compare each of the characters in each string in some kind of loop.
- Insert the following loop:

```
for (int i = 0; i < strlen(si); i++)
{
      if (s1[i] != s2[i]
      {
         printf(“different”);
         return 0;
      }
}
```

```
printf("same");
```

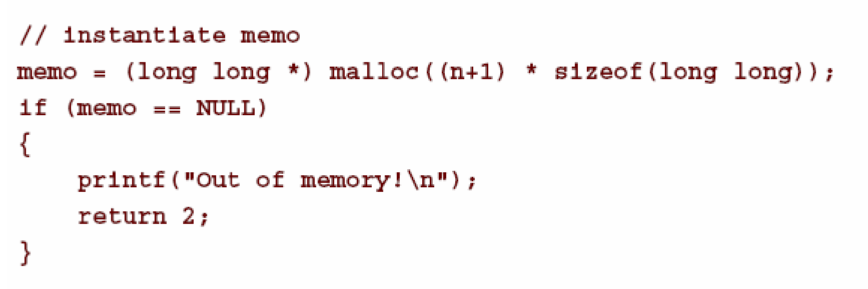
- But now it tells us “foo” and “food” are the same!
- One way to solve this is to insert, before the loop, this piece of code:

```
if (strlen(si) != strlen(s2))
      printf("different");
```

- If the strings have different lengths, we know immediately they’re different. And if they have the same length, then the loop will tell us whether or not they’re the same.

Dynamic Memory Allocation (26:30-37:00)

- Remember our original Fibonacci program? Gosh, that was slow. It was actually exponential in runtime!
- Well, as you might recall, we fixed things a bit by creating an array (memo) to keep track of Fibonacci numbers we’ve already computed. Our new program was a lot faster because it only computed each number in the sequence once.
- The problem with this approach is that we don’t necessarily know how big to make the array in advance
- We want it to be of size n, but n is dynamic (depends on user input), and we can’t declare an array as `memo[n];`
- One way to solve this problem is just to declare a really huge memo that’s large enough to accommodate anything the user could possibly ask for. For instance, we could declare a memo of size 2^{32}.
- This creates another problem. Recall that 2^{32} is about 4 billion. At 4 bytes per integer, we’re talking about a 16 GB array just to compute a Fibonacci number!
- We solve this program by declaring array sizes dynamically using malloc():



```
// instantiate memo
memo = (long long *) malloc((n+1) * sizeof(long long));
if (memo == NULL)
{
    printf("Out of memory!\n");
    return 2;
}
```

- malloc() is a function that sets aside some memory and then returns a pointer to it
- Say you call malloc(4), malloc allocates 4 bytes of memory and returns to you the address of the first one

- In this case, our argument to malloc is (n+1)*sizeof(long long), which is the number of bytes to acoommodate n+1 long longs.
- After we get a pointer back from malloc(), we have to cast it into the desired type so gcc knows how to treat it
- This is essentially an array of n+1 long longs
- So we implement this is fs4.c, and we find that it works pretty fast on 40, 45, 50
- But let's be brave and try fs4 1000000. Oops, segmentation fault!
- Why? Because we tried to allocate too much memory. If malloc() can't give you all the memory you asked for, it gives you nothing.
- Know that whenever you use malloc() to allocate memory, it gets memory from the heap, not the stack. That means it's persistent. Even as functions open and close, this memory continues to be accessible until you explicitly "free" it (as we shall see).

scanf (37:00-47:30)

- Take a look at scanf.c:

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

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

- What are we doing here?
- We prompt the user for a number, and then we read it in and put it in x using scanf().
- scanf() gets two arguments:
    - "%d" – tells scanf() what type of input to expect, notice this is the same format string we use for printf()
    - &x – the *address* of x, tells scanf() where to put the input
- In scanf2.c, we attempt to do this same process with strings, but this version is buggy:

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

    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the \"%s\"!\n", buffer);
}
```

- The problem is that we don't know where buffer is pointing. It's certainly not pointing to any memory we own!

- If we run this, we get a seg fault because we are accessing memory that doesn't belong to us.
- To fix this, we must allocate memory for buffer.
- We can do this by typing `char buffer[4];` or `char * buffer = (char *)malloc(4)`
- The problem with this is that if we ended up entering a string of more than 3 characters, we would be touching memory we don't own.
- If we used `char buffer[4]`, we'd be dealing with memory on the stack, so we'd very quickly run into problems if we tried to touch memory beyond the four allocated bytes, and get a seg fault.
- If we used malloc(), we would be dealing with memory on the heap, so we'd be less likely to get a seg fault. Still, touching memory you haven't allocated is very risky, so it is best just to allocate enough that you won't be likely to extend beyond it.