Topics Covered

Redirection

At the command line, by using **redirection** commands like > or | you can control the input and output of your program.

> - output; print the output of a program to a file instead of stdout (e.g. ./hello > output.txt)
>> - append to an output file instead of writing over data
2> - this is just like the above, instead it will only print out error messages to a file
< - input; use the contents of some file as input to a program e.g. ./hello ¡ input.txt
| - pipe; take the output of one program and use it as input in another

File I/O

So far, we have only been working with stdout (your computer screen) and taking input from the user via get-string, etc. In order for us to have persistent data (data that exists beyond the time our program is running), we need the ability to work with files. Fortunately, we can do so with C. All of the functions we need to operate on files are obtained easily by

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

A full list of the functions that are used for file input/output manipulation, including what parameters each of these functions take, is at http://www.cplusplus.com/reference/clibrary/cstdio/.
Here are some of the big ones:

- fopen(string filename, string method) - opens the file named filename for the reason specified in method (r for reading, w for writing, a for appending), and returns a FILE * (file pointer) to that file.
- fclose(FILE* fp) - closes the file pointed to by fp
- fgetc(FILE* fp) - retrieves the next character in the file pointed to by fp, and returns that character
- fputc(char c, FILE* fp) - writes the character c to the file pointed to by fp. The return value is c, if successful
- fread(void* buffer, int size, int count, FILE* fp) - reads count items of size size from the file pointed to by fp into buffer. The return value is count, if successful
- fwrite(void* buffer, int size, int count, FILE* fp) - writes count items of size size from buffer into the file pointed to by fp. The return value is count, if successful

Here is an example of how to use file I/O (input/output) in code.

```
#include <stdio.h>

int main(void)
{
    // open file "input.txt" in read only mode
    FILE* in = fopen("input.txt", "r");

    // always make sure fopen() doesn't return NULL!
    if(in == NULL)
```

File I/O continued on next page...

```
        return 1;
    else
    {
        // open file "output.txt" in write only mode
        FILE* out = fopen("output.txt", "w");

        // make sure you could open file
        if(out == NULL)
            return 2;

        // get character
        int c = fgetc(in);

        while(c != EOF)
        {
            // write chracter to output file
            fputc(c, out);
            c = fgetc(in);
        }

        // close files to avoid memory leaks!
        fclose(in);
        fclose(out);
    }
}
```

Pointers

Pointers have an awful reputation among beginning programming students because they give you the ability to makesome pretty dramatic mistakes. As long as you are careful with them though you'll be fine. Just remember the following things!

1. **Pointers are Addresses:** A pointer is merely an address in memory. When you dereference a pointer with the dereferencing operator (*) you are simply examining the piece of data that is at that location. For example, in the following code:

   ```
   int* p;
   p = 0x6562AD3E;
   printf("%d\n", *p);
   ```

   All that is happening is that we are looking at the data contained in memory location 0x6562AD3E, interpreting it as an integer, and printing it to the screen. Of course, most people don't know off the top of their heads where everything in memory is, so using memory addresses like the one I've just used is somewhat impractical. That's why C provides the addressing operator &. Here's an example of it.

   ```
   int i = 18;
   int* m;
   ```

```
m = &i;
(*m)++;
```

After the execution of this code, i would have the value 19, instead.

2. **Arrays are secretly just pointers.** After declaring:

```
double averages[40];
```

Any reference to the array name averages is really a pointer to its first element, averages[0]. Likewise, averages' other elements can be accessed by saying *(averages + n), which is equivalent to averages[n].

3. **Pointers give us the power to allocate memory dynamically.** Sometimes, you may not know how much memory you'll actually need when you are programming, and it will be dependent on other factors. Up til now, we've been using only static memory (memory whose size is known by the program at compile time). C gives us the power to allocate memory on the fly (dynamically) as we need it, but we need pointers to do it. To do so, we need to use **malloc()** and **sizeof()**, two tools provided by C to make this happen. Assume that we wait for an input from the user to figure out how many elements an array needs. True, in C99 and making use ofthe CS50 library you can simply do:

```
int arrsize = GetInt();
int arr[arrsize];
```

But it wasn't always possible to do that. You used to have to do this (for simplicity's sake we'll use the CS50library again):

```
int arrsize = GetInt();
int* arr = malloc(sizeof(int) * arrsize);
```

Notice what malloc() does. I am asking malloc() to give me arrsize blocks of contiguous memory, with each block the size of an int(4 bytes apiece). Take care to always make sure that you actually get that memory back though! If malloc() is unable to allocate memory for you, because of an error or because no memory is left, it will return a NULL pointer. Dereferencing a NULL pointer will crash your program. So always do a NULL check! If I pass the NULL check, I can then use this memory in the same way that I did the first example in this section, with one really important exception...

4. **All malloc()'d memory must subsequently be free()'d.** Failure to do so results in what's called a memory leak (which is really bad). When we have finished using a dynamically-allocated piece of memory, we need to give it back to C, in case it needs to use that memory again for something else. If we leak memory by failing to do so, we run the risk of running out of memory and causing our program to crash. It's super easy to free():

```
int x = GetInt();
char* word = malloc(sizeof(char) * x);

// do stuff

free(word);
```

> Just be careful not to double free(), and also be careful only to free() memory that was previously allocated with malloc(), otherwise your risk of a memory leak is still present!

The best way to make sure you are doing this correctly is to use **Valgrind**. It's a program that you will be using on future psets to check to make sure there is no mistakes with your memory (i.e. you forgot to free memory, you dereferenced a null pointer, memory is lost, etc.)

Hexadecimal

Take notes on the example for hexadecimal here!