"Cheat Sheet" - Week 6

CS50 — Fall 2011

Prepared by: Doug Lloyd '09

October 17, 2011

Structures

Don't forget to review structures, found in the Week 4 Cheat Sheet!

`typedef` and `enum`

Using the type name `struct student_t` time after time will inevitably get cumbersome. For this reason, C provides the `typedef` command, which you can use to alias an old type name to a new one. The format is: `typedef <old name> <new name>`. For example:

```
typedef char byte;
typedef struct student_t {
   char *name;
   int year;
   double gpa;
} student;
```

And then, after making those new type definitions, we can use the new names just as we used the old ones:

```
byte letter = 'X';
student john;
student *mary = malloc(sizeof(student));
```

We can also use `typedef` in conjunction with another C keyword, `enum`, to create our own enumerated types. Enumerated types are used to create a new type when that type can only take on a specific set of values. For example, the oft-used `bool` type can be `enum`'d as follows:

```
typedef enum { false, true } bool;
```

And that creates a new enumerated type called `bool` which can take on the values `false` and `true`. Underneath the hood, C assigns each of the values an enumerated type can take on a sequential integer starting from 0. Hence why we chose to put `false` before `true`, since in C, 0 is equivalent to false, and everything else is true. The "counting from 0" can be overriden quite easily as well:

```
typedef enum { CABOT = 1, CURRIER, PFOHO } quad_houses;
```

File I/O

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. Just `#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/*. Take a look! Here's a simple program that takes 2 command line arguments, a source file (to read from) and a destination file (to write to), and assuming each is a simple text file, copies the first file into the second!

```
#include <stdio.h>
int main(int argc, char *argv[]) {
   if(argc != 3)
      return 1;
```

```
   FILE *src = fopen(argv[1], "r"); // open argv[1] for reading
   FILE *dest = fopen(argv[2], "w"); // open argv[2] for writing
   char ch;
   while((ch = fgetc(src)) != EOF) // read one char at a time until file's end
      fputc(ch, dest);
   fclose(src); // close the source (so no memory leak!)
   fclose(dest); // close the destination
   return 0;
}
```

Linked Lists

Linked lists come in singly- and doubly-linked flavors. This sheet focuses only on singly-linked lists, but the concepts herein can easily be generalized to doubly-linked lists. Linked lists give us a new data structure for holding, arranging, searching for, sorting, deleting, and inserting values. Previously, all we had was the array! Linked lists are a very special structure, where the fields of the structure are some data and a pointer to another structure of the same type. In code form:

```
typedef struct _sllist {
   int data;
   struct _sllist *next;
} sllist;
```

Let's create a singly linked list using a few of these nodes!

```
int main() {
   sllist first;
   sllist second;
   sllist *third = malloc(sizeof(sllist));
   first.data = 5;
   second.data = 6;
   third->data = 8;
   first.next = &second;
   second.next = third;
   third->next = NULL;
   ...
   free(third);
   return 0;
}
```

Now let's write a function to traverse the list and print out all the values! This function will take a pointer to the first node in the list (usually called the *head* of the list), and will continue printing until it hits a NULL value (which we typically use to represent the *tail* of the list). Let's do it both iteratively and recursively!

```
void printList_I(sllist *head) {
   while(head != NULL) {
      printf("%d ", head->data);
      head = head->next;
   }
   return;
}
```

```
void printList_R(sllist *head) {
   if(head == NULL)
      return;
   else {
      printf("%d ", head->data);
      printList_R(head->next);
   }
   return;
}
```

And above, we'd just replace the `...` line in the previously-defined `main()` with `printList_I(&first)` or `printList_R(&first)`, as appropriate. The things you need to know how to do with a linked list are:

- Create a list.
- Insert an item after another item already in the list.
- Insert an item before another item already in the list. (Doubly-linked lists only)
- Add an item to the end of a list.
- Delete a single node from the list.
- Delete the entire linked list.

Try playing around with the sample code a bit to see if you can get these things to work! (Hint: The last one must be done recursively with a singly-linked list. Think about it!)