"Cheat Sheet" - Week 5

CS50 — Fall 2012

Prepared by: Doug Lloyd '09

October 15, 2012

Structures

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

typedef

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));
```

GDB

GDB stands for the "GNU Debugger", a tool built into the Appliance (and, indeed, many UNIX-flavored installations) that we can use to debug our code. Used correctly, we can use it to "freeze" execution at a given point in our program and step through it line-by-line. Then, using some of GDB's built-in commands, we can learn lots about the state of our program as we step through it–that information is frequently useful to find out where things are going wrong! Type `gdb <program name>` at the command line to get started, then use these commands[1] once inside the GDB program:

- `(b)reak <function name>` - "breaks" execution of the program at the specified point, so that from there you'll have to step through your code. Basically used so that you don't step through all of your code if the problem is isolated in a particular function.
- `(r)un [args]` - after breaking, you'll want to run your program–perhaps with command line arguments.
- `(p) <variable name>` - prints out the value of the specified variable
- `info locals` - prints out the names and values of all local variables in the current function
- `(n)ext` and `(s)tep` - respectively, advance to the next block or line of code. `s` sometimes will dive a little more deeply than you want (for example, if you `s` in a call to, say, `strlen`, you'll start stepping through the code for `strlen` one line at a time), so just make sure you're where you want to be!
- `bt` (backtrace) - tells you the chain of function calls that got you to the current point in the program (aka, looks down the stack)
- `(q)uit` - quits GDB

[1] A far more comprehensive and detailed list can be found at http://www.stanford.edu/class/cs107/other/gdbrefcard.pdf

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 are some of the big ones:

- `fopen(char *filename, char *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

So let's see about putting some of these functions to use. UNIX systems have a built in command line command called `more` which opens a file specified at the command line and prints its entire contents to the terminal screen. Let's write code that replicates `more`.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   // ensure proper command line arguments
   if(argc != 2) {
      printf("Usage: ./more <filename>\n");
      return 1;
   }

   // open the file and make sure we're successful
   FILE *fp = fopen(argv[1], "r");
   if(fp == NULL) {
      printf("Error: Could not open %s for reading\n", argv[1]);
      return 2;
   }

   // let's do this one character at a time
   char c;
   while((c = fgetc(fp)) != EOF)
      printf("%c", c);

   // and we're done!
   fclose(fp);
   return 0;
}
```

Redirection and Pipes

Instead of prompting the user for input each time, or only writing out messages to the terminal, we can direct our programs to read from, and write to, files–without needing to write any additional code. Instead, at the command line, we can use `<`, `>`, `>>`, and `|`. To direct your program to read input from another file, use `<`:

```
jharvard@appliance (~/pset3): ./scramble < words
```

That oughta rack up some points. To direct your program to output all your `printf`s to a file, instead of the terminal screen, use `>` or `>>` (the latter if you want to append to, instead of overwrite, the specified file):

```
jharvard@appliance (~/pset2): ./vigenere FOOBAR > cipher.txt
jharvard@appliance (~/pset2): ./vigenere BARFOO >> cipher.txt
```

You can even combine these together. Assume we modified mario.c slightly to use `atoi` and `GetString()`, instead of `GetInt()`. This will read a height (as a string) stored in a text file called height.txt, and will print out the pyramid to a file called pyramid.txt:

```
jharvard@appliance (~/pset1): ./mario < height.txt > pyramid.txt
```

`|` (vertical bar, or pipe) has similar uses. Check it out on your own!