"Cheat Sheet" - Week 2

CS50 — Fall 2011

Prepared by: Doug Lloyd '09

September 19, 2011

Functions

Function declarations have three parts: a **type**, a **name**, and a comma-separated, **argument list**, each argument having a type and a name. They end with a semicolon. The following are examples of valid function declarations:

```
int add_two_nums(int arg1, int arg2);
char first_letter_is(string word);
void print_instructions();
```

The following are examples of invalid function declarations.

```
int (char letter1, char letter2);
simple_function(int input);
int higher_number(int num1; int num2);
```

A function defintion must occur *after* and *separately from* the function declaration. The beginning of the function defintion should match, perfectly, the function declaration (except for the semicolon). As an example, going off of the `add_two_nums()` function declared above:

```
int add_two_nums(int arg1, int arg2) {
   int sum = 0;
   sum = arg1 + arg2;
   return sum;
}
```

Directives

The directives (or, macros) that you encounter most are `#include` and `#define`. When the compiler sees `#include`, it essentially copies the contents of the file you list into your program's object code. When the compiler sees `#define`, it goes through and substitutes any instances of the "symbol" you use to represent the "magic number" (which could also be a letter, a word, or even a small function!) with that "magic number". For instance:

```
#include <cs50.h>
#define YEAR 2011
```

The former would paste the entire contents of `cs50.h` atop the .c file that contains the `#include`. The latter would literally replace all instances of `YEAR` with "2011" in your object code. Make sure NOT to put semicolons at the end of your `#define`s!

Scope

A variable's scope is a characteristic of how visible that variable is to other functions. A variable's scope can be global, or local. In the below example, `a` is global, and can be used and manipulated by all functions. `b` is local to `main()` and `c` is local to `f1()`. `d` is local only within the context of the `for` loop in `main()`. That is, `d` means nothing anywhere else but while the program is running through that loop.

```
void f1();

char a = 'i';

int main(void) {
   int b = 10;
   for(int d = 1; d <= b; d++)
      f1();
   return 0;
}

void f1() {
   char c = 'h';
   printf("%c%c\n", a, c);
}
```

Arrays

Arrays are collections of variables of a like type. An array is declared with the following syntax:

```
int student_grades[13];
```

This would create an array of 13 integers. The array name is `student_grades`. I can access individual elements of the array easily:

```
student_grades[5] = 98;
student_grades[10] = 85;
```

C will let you go "out of bounds" on your arrays. In an array of size n, the indexes you can access are in the range $\{0, \ldots, n-1\}$. In this case, then, I can access elements $0-12$ of `student_grades` without running into any trouble. Arrays can also be multidimensional, and individual elements can be accessed in an identical manner:

```
char tic_tac_toe[3][3];
tic_tac_toe[1][2] = 'X';

double hypercube[8][10][6][20];
hypercube[4][9][0][15] = 6.2569;
```

Using `argc` and `argv`

`argc` is an integer variable, provided by `main()` that tells you how many command-line arguments were inputted. `argv` is an array of `strings` (or, `char *`s), that contain the actual command-line arguments themselves. You can manipulate both, e.g.:

```
./this class is cool
```

Here, `argc` would be 4, `argv[0]` would be `./this`, and `argv[3]` would be `cool`.