Computer Science 50
Fall 2009
Scribe Notes

Week 2 Wednesday: September 16, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 2 Wednesday: September 16, 2009
Andrew Sellergren

1 Announcements (0:00–11:00)

- This is CS 50.
- If you did the Hacker Edition of Problem Set 1, be sure to return your sensor board after class and write your name on the slip of paper.
- 0 new handouts.
- Lunch with David! If you're interested in joinining David and a handful of others in an opportunity to make a large class much more intimate, then go here to RSVP. Lunch with David will be held Fridays at 1:15 PM. If you've already RSVP'ed, we'll be in contact with you shortly regarding details. There's also an upcoming Dinner with David at Mather as part of their Senior Common Room. Go here to RSVP for this one-time event!
- Feel free to call David by his first name. Or Mocking CS 50 Czar.[1]
- Section assignments have been sent out. If you didn't receive yours or you have a conflict with the time you were assigned, please get in touch with us via e-mail.
- Check the website for the Office Hours schedule. Before you come to Office Hours with a question like "Where do I begin?" on the problem set, be sure to watch Marta's walkthrough. We do move quickly in this course, but once you start really plugging away at it (learning by *doing*), we think you'll be surprised at how capable you are.
- The CS 50 Map has been improved! Thanks to the MarkerClusterer, the map's load time has been significantly decreased (adding 350 cartoon markers to the map takes a lot of JavaScript). In addition, we addressed some problems which were causing to deny our geocoding requests. Finally, we made sure that if multiple people were from the same location, all of their names would appear.

2 Some More Bugs (11:00–50:00)

- Some of you may have already encountered the problem of rounding floating-point values. Let's take a look at `imprecision.c` to demonstrate this problem:

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

int
main()
```

[1] Actually, he really likes it if you call him Malan. No, really, don't. You wouldn't like him when he's angry.

```
{
    float f = 0.01;
    printf("%.10f\n", f);

    int i = round(f * 100);
    printf("%d\n", i);
}
```

If we run this program, we see that even though we've perfectly specified our floating-point value as 0.01, that value is being stored as something imprecise like 0.0099999998. So if we take this value and multiply by 100 (to move the decimal place two to the left), we're going to get 0.99999998. Then if we assign that to an `int`, it will become 0. What's the solution? Use the function `round` declared in the `math.h` library. Don't forget to link it at compile time! The `man` page will tell you that you use the flag `-lm` to do so.

- To read more about the `math.h` library, check out the CPP Reference. We've taken the CPP manual and stripped out all the references to C++ to avoid any confusion.
- Let's take a look at another subtle bug in `buggy3.c`:

```
/**************************************************************************
 * buggy3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should swap two variables' values, but doesn't!
 * Can you find the bug?
 **************************************************************************/

#include <stdio.h>


// function prototype
void swap(int, int);


int
main(int argc, char *argv[])
{
    int x = 1;
    int y = 2;

    printf("x is %d\n", x);
```

```
    printf("y is %d\n", y);
    printf("Swapping...\n");
    swap(x, y);
    printf("Swapped!\n");
    printf("x is %d\n", x);
    printf("y is %d\n", y);
}


/*
 * void
 * swap(int a, int b)
 *
 * Swap arguments' values.
 */

void
swap(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}
```

Before we go through this program line by line, step back a moment and think about what the program is supposed to do. Basically, it's just demonstrating the use of a function called `swap`, which swaps the values of two variables.

- Let's compile the program and see if it works. We can use the `make` command so that the `-lcs50` and `-lm` flags are automatically included. But when we run the program, we get the following output:

```
x is 1
y is 2
Swapping...
Swapped!
x is 1
y is 2
```

What's going on here? If we take a closer look at `swap`, we can see that it's logically sound. We're using non-descriptive variable names like `a`, `b`, and `tmp`, but that's okay because the function is really rather short and simple. Note, of course, we need a third variable because if we just assign

the value of `b` to `a`, then we lose track of the value of `a`. So we just need a place to store it while we make the swap.

- So what exactly is wrong? As someone pointed out, the function is not returning any value because its return type is `void`. But that's not really the problem. And neither is the fact that we've named the variables `x` and `y` in our `main` method and `a` and `b` in our `swap` function. The variable names `a` and `b` are really just placeholders for whatever arguments we've passed in. Consider that whoever wrote `printf` isn't aware of what variable names you used in your own script, but has his own name for variables that are used within the `printf` function itself. So `a` and `b` just stand in for `x` and `y`.
- What's really happening when we call `swap`? Are we passing in `x` and `y` themselves? No, in fact, we're passing in copies of those variables. So while `swap` successfully swaps the values of `a` and `b`, it doesn't actually have any effect on the values of `x` and `y`.
- In computer science speak, we say that functions like `swap` have their own *scope*. Because `x` and `y` exist in a different context, the function can't alter them directly—at least the way we've written it here. We'll discuss a solution to this shortly.
- We can demonstrate this idea of scope with `buggy4.c`:

```
/**************************************************************************
 * buggy4.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should increment a variable, but doesn't!
 * Can you find the bug?
 **************************************************************************/

#include <stdio.h>


// function prototype
void increment();


int
main(int argc, char *argv[])
{
    int x = 1;
    printf("x is now %d\n", x);
```

```
    printf("Incrementing...\n");
    increment();
    printf("Incremented!\n");
    printf("x is now %d\n", x);
}


/*
 * void
 * increment()
 *
 * Tries to increment x.
 */

void
increment()
{
    x++;
}
```

If we try to compile this, we get the following errors:

```
buggy4.c: In function 'increment':
buggy4.c:40: error: 'x' undeclared (first use in this function)
buggy4.c:40: error: (Each undeclared identifier is reported only once
buggy4.c:40: error: for each function it appears in.)
```

Although we declare `x` as an `int` in our `main` method, we can't reference it in `increment` because it has a different scope. `increment` doesn't know anything about where `x` is stored in memory because it has its own separate chunk of memory to work with.

- How can we at least get this program to compile? Well we can declare `int x` in `increment`. Be careful, though, we need to initialize it with a value. If we don't, then we'll be incrementing whatever junk is already stored in memory at that location from a previous use. So write `int x = 0`. Now when we compile and run it, we get the following output:

```
x is now 1
Incrementing...
Incremented!
x is now 1
```

Obviously, that didn't really solve the problem. What if we take the approach of declaring `x` outside the scope of both `main` and `increment`? Let's put the `int x = 0` line right below our declaration of the `increment`

function and recompile. Now it works! But I'm not even going to bother to show you those lines of code because it's bad practice! In general, we want to avoid the use of these so-called *global variables*. When your code is spread across multiple files and hundreds of lines, you're liable to forget the names of global variables and you'll end up clobbering them without knowing it. And, trust us, there are other reasons why you should avoid global variables, but we won't go into them for now.

- What about that declaration of the `increment` function?

  ```
  void increment();
  ```

  If we get rid of this line of code, we get the following errors at compile time:

  ```
  buggy4.c: In function 'main':
  buggy4.c:19: warning: implicit declaration of function 'increment'
  buggy4.c: At top level:
  buggy4.c:34: warning: conflicting types for 'increment'
  buggy4.c:19: warning: previous implicit declaration of 'increment' was here
  ```

  The compiler is mad because we're calling `increment` before it's been defined. We can actually fix this simply by placing the definition of `increment` above the definition of `main`. This is a little sloppy style-wise, though, since someone who's reading your code probably wants to see the `main` method first. The better solution is to declare the function by stating its return type, its name, and its argument types at the top of the file. This is, unfortunately, one of the downsides of C that has been done away with in most modern programming languages.

- To shed more light on this problem of scope, take a look at `buggy5.c`:

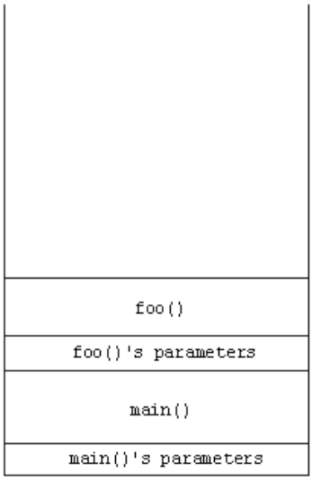
  ```
  /**************************************************************************
   * buggy5.c
   *
   * Computer Science 50
   * David J. Malan
   *
   * Should increment a variable, but doesn't!
   * Can you find the bug?
   **************************************************************************/

  #include <stdio.h>


  // global variable
  int x;
  ```

```
// function prototype
void increment();


int
main(int argc, char *argv[])
{
    printf("x is now %d\n", x);
    printf("Initializing...\n");
    x = 1;
    printf("Initialized!\n");
    printf("x is now %d\n", x);
    printf("Incrementing...\n");
    increment();
    printf("Incremented!\n");
    printf("x is now %d\n", x);
}


/*
 * void
 * increment()
 *
 * Increments x.
 */

void
increment()
{
    int x = 10;
    x++;
}
```

When we run this program, it prints the value 1 for x both times. Why is that? When `increment` is called, we declare a variable `int x`. When we execute the statement `x++`, then, we're incrementing this newly declared variable rather than the one which is global in scope. This reuse of a variable name in different scopes is called *shadowing* and is perfectly fine as long as you keep track of which version of the variable is being manipulated at any given time.

- What's actually going on in memory when we deal with these variables? The FAS servers have a large chunk of memory called their RAM. When your program runs, it takes memory from the very "bottom" of the memory bank and sets it aside to store `main`'s parameters. We can visualize

this below:



Next in memory are stored the variables which are declared within the scope of `main`. Then, if there are any other functions we call such as `foo`, we tack on the parameters that are passed to `foo` as well as `foo`'s *local variables*—the variables declared within its scope. This is a very brief look at what is known in computer science as the *stack*, which is simply a way of conceptualizing a computer's RAM. As you can see from this visualization, each function has its own *frame*, or chunk of memory, and doesn't have access to any other function's frame. Global variables are actually stored at the "top" of memory, called *heap*, along with what's called the text segment of your program—its representation in binary after being compiled.

- As one astute student suggested, we could fix our problem of scope by implementing a function that has an actual return value. Take a look at `return1.c` to see this in action:

```
/**************************************************************************
 * return1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Increments a variable.
 *
 * Demonstrates use of parameter and return value.
 **************************************************************************/

#include <stdio.h>
```

```
// function prototype
int increment(int);


int
main(int argc, char *argv[])
{
    int x = 1;
    printf("x is now %d\n", x);
    printf("Incrementing...\n");
    x = increment(x);
    printf("Incremented!\n");
    printf("x is now %d\n", x);
}


/*
 * int
 * increment(int a)
 *
 * Returns argument plus one.
 */

int
increment(int a)
{
    return a + 1;
}
```

- Slightly more interesting is `return2.c` which implements a function that cubes its input:

```
/****************************************************************************
 * return2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Cubes a variable.
 *
 * Demonstrates use of parameter and return value.
 ****************************************************************************/

#include <stdio.h>
```

```
// function prototype
int cube(int);


int
main(int argc, char *argv[])
{
    int x = 1;
    printf("x is now %d\n", x);
    printf("Cubing...\n");
    x = cube(x);
    printf("Cubed!\n");
    printf("x is now %d\n", x);
}


/*
 * int
 * cube(int a)
 *
 * Cubes argument.
 */

int
cube(int a)
{
    return a * a * a;
}
```

Hopefully you're getting used to this kind of programmatic flow whereby we factor out and abstract away certain operations into separate functions. As before, we give the input of `cube` a very uninteresting name since it's short-lived. We do the math that's necessary and then return the value which will overwrite the value of `x`.

3 New Section! (50:00–50:00)

- I'm really bad at dividing these notes into sections. So here's one just to help break it up a little bit.

4 Arrays (50:00–60:00)

- Let's say we want to write a program to store our grades from all our semesters at Harvard. We've familiarized ourselves with the concept of

a loop, so it should seem obvious that we'd want to implement one here. But how will be dynamically create variables to store the grades we take as input from the user? We'll use arrays!

- Recall from Scratch the variable type lists. This is simply the Scratch implementation of arrays. In other contexts, arrays are called vectors.
- Check out this fantastic tutorial on arrays in C.
- So let's say we want to declare an array to store 32 courses. The syntax for this is as follows:

  ```
  int courses[32];
  ```

  This simply asks the compiler for 32 `int`'s contiguous in memory. Now, to assign values to this variable, we write `courses[0]`, `courses[1]`, `courses[2]`, etc.

- Take a look at `array1.c` below:

```
/**************************************************************************
 * array1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Computes a student's average across 2 quizzes.
 *
 * Demonstrates use of an array, a constant, and rounding.
 **************************************************************************/

#include <cs50.h>
#include <stdio.h>


// number of quizzes per term
#define QUIZZES 2


int
main(int argc, char *argv[])
{
    float grades[QUIZZES], sum;
    int average, i;

    // ask user for grades
    printf("\nWhat were your quiz scores?\n\n");
```

```
    for (i = 0; i < QUIZZES; i++)
    {
        printf("Quiz #%d of %d: ", i+1, QUIZZES);
        grades[i] = GetFloat();
    }

    // compute average
    sum = 0;
    for (i = 0; i < QUIZZES; i++)
        sum += grades[i];
    average = (int) (sum / QUIZZES + 0.5);

    // report average
    printf("\nYour average is: %d\n\n", average);
}
```

As you might have figured out from reading the code yourself, this program takes as input a certain number of quiz grades, stores them in an array, and then computes their average. Notice that instead of a hard-coded numeric value, we use `QUIZZES` to specify the number of inputs to store. This is to avoid the problem of *magic numbers*. Magic numbers are those that appear in a program without any particular justification. Not to mention, they make the program particularly inflexible. What if later on we wish to change the number of quizzes that this program averages? It would be a pain to go through all the lines of code and replace a hard-coded numeric value. In the above, however, we need only change how the constant `QUIZZES` is defined at the top with the `#define` pre-processor directive. Another advantage of this is optimizations which the compiler can perform. By convention, constants are written in all capital letters to distinguish them from variables.

- Notice that we can consecutively declare multiple variables of the same type by separating them with variables. Realize that arrays are always zero-indexed.
- What's with the addition of the 0.5? Think it through. It's a quick trick to implement rounding without using the `round` function. `array2.c` is equivalent functionally to `array1.c` except it explicitly invokes the `round` function.

5 Strings as Arrays (60:00–70:00)

- We said before that in C a string is actually implemented as a `char *`. We'll get more into that later in the course, but for now let's consider another way of conceptualizing a string: as an array. As it turns out, a string can be accessed as an array of `char`'s.

- Take a look at a `string1.c`:

```
/****************************************************************************
 * string1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a given string one character per line.
 *
 * Demonstrates strings as arrays of chars and use of strlen.
 ****************************************************************************/

#include <cs50.h>
#include <stdio.h>
#include <string.h>


int
main(int argc, char *argv[])
{
    char c;
    int i;
    string s;

    // get line of text
    s = GetString();

    // print string, one character per line
    if (s != NULL)
    {
        for (i = 0; i < strlen(s); i++)
        {
            c = s[i];
            printf("%c\n", c);
        }
    }
}
```

  Here we're taking a string as user input and then printing it out one letter at a time. The `s != NULL` is actually an error-checking mechanism. If `GetString()` returns this *sentinel* value of `NULL`, we won't execute this loop at all. This is because if an error occurs, we don't want to even attempt to access the string as an array. We'll see why in a moment.

- Hopefully it's obvious, but the `strlen()` function takes a string as input

and returns its length. As it turns out, there's a special value at the end of all strings: `\0`, which represents `00000000`.

- In `string2.c`, we make a single optimization in our design:

```
/****************************************************************************
 * string2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a given string one character per line.
 *
 * Demonstrates strings as arrays of chars with slight optimization.
 ****************************************************************************/

#include <cs50.h>
#include <stdio.h>
#include <string.h>


int
main(int argc, char *argv[])
{
    int i, n;
    string s;

    // get line of text
    s = GetString();

    // print string, one character per line
    if (s != NULL)
    {
        for (i = 0, n = strlen(s); i < n; i++)
        {
            printf("%c\n", s[i]);
        }
    }
}
```

In our `for` loop, we have two initializations. Why did we assign the return value of `strlen(s)` to another variable? This approach prevents a function call every single time the loop executes. Think about it: is the string's length changing each time the loop executes? No. Does it make sense, then, to calculate its length every time, most likely by iterating over every character in the string? No. It's obviously a waste of resources to call `strlen()` more than once.

- `capitalize.c` is a program which brings together a lot of the concepts we've touched upon so far:

```
/****************************************************************************
 * capitalize.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Capitalizes a given string.
 *
 * Demonstrates casting and iteration over strings as arrays of chars.
 ****************************************************************************/

#include <cs50.h>
#include <stdio.h>
#include <string.h>


int
main(int argc, char *argv[])
{
    int i, n;
    string s;

    // get line of text
    s = GetString();

    // capitalize text
    for (i = 0, n = strlen(s); i < n; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
            printf("%c", s[i] - ('a' - 'A'));
        else
            printf("%c", s[i]);
    }
    printf("\n");
}
```

Recall that in ASCII, lowercase `a` is 97 and uppercase `A` is 65. So to capitalize a letter, we simply subtract the difference between those numbers. We could, as before, store the value of `'a' - 'A'`, but the compiler will probably actually do that for us since it can recognize simple arithmetic like this. In any case, writing it this way is much more clear than inputting the magic number of 32.

- If you're feeling like this is moving too fast or you just want some extra resources, check out the How Stuff Works tutorial and the CPP Reference.
- Let's examine that copy-paste job that we've been putting at the top of every program we've written so far:

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

  As it turns out, `main` is its own function just like any custom one we might write. `int argc` and `char *argv[]` are, in fact, two arguments that we specify in its function definition. How do we provide these arguments? From the command line!
- But you've been doing this all along. Recall that whenever you've executed the `gcc` command, you've provided it with the name of the file you want to compile as a command-line argument.
- Take a look at `argv1.c`:

```
/**************************************************************************
 * argv1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints command-line arguments, one per line.
 *
 * Demonstrates use of argv.
 **************************************************************************/

#include <stdio.h>


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

    // print arguments
    printf("\n");
    for (i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    printf("\n");
}
```

What does this program do? It actually prints out the command-line arguments that we provide it. `argv[]` is not just an array, but an array

of strings, or an array of arrays. More than that, it is an array of the command-line arguments that have been provided to the program. `argc` is the count of those arguments.

- `argv2.c` takes it one step further:

```
/**************************************************************************
 * argv2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints command-line arguments, one character per line.
 *
 * Demonstrates argv as a two-dimensional array.
 **************************************************************************/

#include <stdio.h>
#include <string.h>


int
main(int argc, char *argv[])
{
    int i, j, n;

    // print arguments
    printf("\n");
    for (i = 0; i < argc; i++)
    {
        for (j = 0, n = strlen(argv[i]); j < n; j++)
            printf("%c\n", argv[i][j]);
        printf("\n");
    }
}
```

Now we're iterating over not only all of the command-line arguments, but over each of the arguments themselves and printing them out one character at a time. We access each of these characters using the following syntax:

`argv[i][j]`

This gives the j^{th} character of the i^{th} argument. `argv[]`, more properly speaking, is a two-dimensional array.

- Soon we'll be peeking under the hood at CS 50's library, which provides a plethora of functions for your disposal. One thing worth mentioning is

that this library has intentional memory leaks built into it. That is, it allocates memory to you that it never asks you to return. We'll explain more later.

- Where are we going with all this? For starters, the world of cryptography in Problem Set 2! You'll implement Caesar's cipher by leveraging the fact that strings are arrays of characters.
- To conclude, ponder the fact that The Simpsons is the greatest television show of all time.