Computer Science 50
Fall 2010
Scribe Notes

Week 2 Wednesday: September 15, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 2 Wednesday: September 15, 2010
Andrew Sellergren

1 Announcements and Demos (0:00–10:00)

- This is CS50.
- Inspired by a note from a student, check out this scene from the 1969 film The Computer Wore Tennis Shoes. The rest of the movie actually has very little to do with computer science.
- Sectioning is taking place now. The FAS sectioning tool will be active later today.
- Take advantage of office hours, the schedule for which is here. Physical office hours are held in Science Center B14. When you walk in, write your name on the whiteboard and we'll get to you one by one. The purpose of office hours is to get you unstuck from a problem that you might otherwise waste several hours figuring out on your own. There is a time crunch, however, so one you get unstuck, the teaching fellow that is helping you may move on to the next student in line. If you'd like to spend more time hashing out concepts, feel free to reach out to your teaching fellow or David for a private appointment.
- The course's bulletin board is now active. You can browse other students' questions and post your own, as well. If you're sharing a substantial amount of code or the question is otherwise person, you can mark it as private. Alternatively, you can e-mail help@cs50.net. By default, your posts will be anonymized: only the staff will be able to see who you are.
- Another option available to you is virtual office hours. By logging into the Virtual Terminal Room at the designated times, you'll be able to chat and share your screen with teaching fellows who can hopefully resolve your minor issues without your having to drag yourself down to the basement of the Science Center.

2 Some Common Mistakes (10:00–18:00)

- Take a look at the following lines of code which we've been seeing a lot during office hours:

```
printf("Input a number: ");
GetInt();
```

  The problem with this is that the return value of `GetInt()` is not being stored. We can demonstrate this with volunteers, one of whom represents `GetInt()`, one of whom represents the user, and one of whom represents a variable. `GetInt()` hands a piece of paper to the user and asks her for a number. The user writes the number down on that piece of paper and passes it back to our program. But if our variable isn't there to catch the piece of paper, it simply falls to the floor unused. Only when the variable

is present to catch the piece of paper are we able to use the value in the rest of our program.

- See if you can identify the addition problem with the lines below:

```
printf("Input a number: ");
{
    GetInt();
}
```

The curly braces aren't necessary here, as they are only necessary to encapsulate loops, conditions, and functions. This code actually will compile, but it's definitely not good style.

- In this example, the semicolon is misplaced:

```
for (int i = 0; i < 100; i++);
{
    // do something
}
```

The effect will be for the code in the curly braces to execute only once as the `for` loop has been terminated prematurely by the semicolon.

3 Functions and Scope (18:00–60:00)

- You can think of a function as a black box. If it was written by someone else, you don't need to know how exactly it does what it does, you just need to know what inputs to give it and what outputs it will return.

3.1 `return1.c`

- Take a look at our first foray into writing our own functions:

```
/**************************************************************************
 * 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 a);


int
main(void)
{
    int x = 2;
    printf("x is now %d\n", x);
    printf("Incrementing...\n");
    x = increment(x);
    printf("Incremented!\n");
    printf("x is now %d\n", x);
}


/*
 * Returns argument plus one.
 */

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

At the top of the file, we have to declare our function name just like we declare variables. This is called a function *prototype*. Notice that it ends with a semicolon.

- Any time you find yourself tempted to copy and paste several lines of code over and over again, chances are those lines of code are good candidates for being factored out into a separate function. In computer science speak, this is called *hierarchical decomposition*.
- You should give meaningful names to your functions just as you do your variables.
- The function's inputs, also known as parameters or arguments, are specified in the parentheses after its name. In this case, our input doesn't need a meaningful name because it is only being used very temporarily and very transparently in the function code, much like an iterator variable in a loop.
- Be careful, as always, with variable types. If you specify that your function returns an `int`, make sure that you actually return one or you may run into compiler errors or typecasting bugs.

- Of course, this function is not the most interesting since it can easily be replaced with the `++` operator, but it illustrates at least our capability to abstract away certain tasks.
- Take a look at what happens when we try to compile if we don't include the function prototype at the top of our file:

  ```
  cc1: warnings being treated as errors
  return1.c: In function ‘main’:
  return1.c:21: error: implicit declaration of function ‘increment’
  make: *** [return1] Error 1
  ```

  Notice that GCC points us to line 21, when `increment` is first called, as the source of the error. When `return1` is being compiled, the compiler looks first for the function named `main`. Since `main` is defined before `increment`, it will throw an error as soon as `increment` is called because there is no definition for it yet. To fix this, we can either put the definition of `increment` before that of `main` or we can write a function prototype, which declares the function but does not define it, at the top of the file. The latter solution is preferable because it makes our file more readable and it works even in scenarios in which functions call each other and correctly ordering them top to bottom is not possible.
- Question: how can you name the function input `a` in the definition but pass it as `x` when you call it within `main`? Variable names actually belong, so to speak, to the functions that are using them. In this case, `x` belongs to `main` and `a` belongs to `increment` though they have the same variable. Underneath the hood, 8 bytes of memory are allocated for them, 4 bytes for each copy of the variable. This can cause interesting bugs as we'll see in a moment.

3.2 `return2.c`

- `increment` was a pretty useless function, but you can see how we might build off the idea to produce something a little more interesting:

```
/**************************************************************************
 * 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 a);


int
main(void)
{
    int x = 2;
    printf("x is now %d\n", x);
    printf("Cubing...\n");
    x = cube(x);
    printf("Cubed!\n");
    printf("x is now %d\n", x);
}


/*
 * Cubes argument.
 */

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

The `cube` function is a little more compelling since it will save us from writing `x * x * x` whenever we need to cube something.

3.3 Scope

3.3.1 `buggy3.c`

- As we mentioned earlier, variables belong to the functions that manipulate them. This is the concept of *scope*. Variable scope can introduce some subtle bugs, as we'll soon see.
- How would we go about swapping the values of two variables? Realize that we can't accomplish this without using a temporary variable. For example, the following lines of code won't work:

```
int x = 1;
int y = 2;
```

```
x = y;
y = x;
```

Here, both variables will end up having the value 2 because the value of `x` is clobbered. We can fix this like so:

```
int x = 1;
int y = 2;
int tmp = x;

x = y;
y = tmp;
```

This, as you can see, is very similar to our implementation of `swap` 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 a, int b);


int
main(void)
{
    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);
```

```
}


/*
 * Swap arguments' values.
 */

void
swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

The `swap` function has a return type of `void` because we actually don't need it to return anything at all.

- So everything looks fine, 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
```

- Hmm, let's try adding some `printf` statements to `swap` so we can examine the values of `a` and `b`:

```
/*
 * Swap arguments' values.
 */

void
swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
    printf("a=%d\n", a);
    printf("b=%d\n", b);
}
```

Now we get the following output:

```
x is 1
y is 2
Swapping...
a=2
b=1
Swapped!
x is 1
y is 2
```

- 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.
- Question: why isn't `tmp` mentioned in the function prototype for `swap`? You only need to specify the function's return type, name, and arguments, **not** its local variables.
- Question: do curly braces define scope? Yes and no. It's a good rule of thumb that variables declared within curly braces can only be used within those same curly braces.

3.4 Global Variables

3.4.1 `global.c`

- One way to fix this problem of scope is to use *global variables*. These variables are unique in that they can be referenced by any function in a program. Take a look at `global.c` to see how they are used:

```
/**************************************************************************
 * global.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Increments variables.
 *
 * Demonstrates use of global variable and issue of scope.
 **************************************************************************/

#include <stdio.h>
```

```
// global variable
int x;

// function prototype
void increment(void);


int
main(void)
{
    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);
}


/*
 * Increments x.
 */

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

Here we declare the variable `x` outside the scope of both `increment` and `main`, right above the function protoype. This solves our problem in that `x` can be referenced in any of the functions in this file, but it's not the cure-all that it might seem. Global variables are generally considered bad style because they're hard to keep track of. If you're using code from other libraries, for example, the names of your global variables might inadvertently conflict with the names of library variables and introduce subtle bugs into your program.

- The `increment` function now returns `void` and takes `void` as arguments because it can reference `x` without it having been passed.

3.5 buggy5.c

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

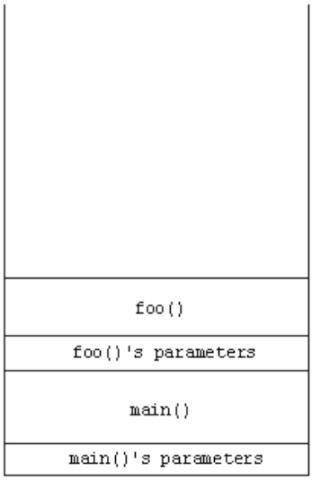
```
/**************************************************************************
 * buggy5.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Should increment a variable, but doesn't!
 * Can you find the bug?
 **************************************************************************/

#include <stdio.h>


// global variable
int x;

// function prototype
void increment(void);


int
main(void)
{
    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);
}


/*
 * Increments x.
 */

void
increment(void)
{
```

```
    int x = 10;
    x++;
}
```

- As an aside, we are able to use `printf` without declaring it in our file because we include `stdio.h`. `stdio.h` is actually header file that contains nothing but function prototypes so the compiler will know about these other functions.
- 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.

3.6 The Stack

- What's actually going on in memory when we deal with these variables? Variables are stored in RAM, a temporary kind of memory, as opposed to on the hard disk, a more permanent kind of memory. We can visualize a computer's RAM like so:



As a matter of convention, `main`'s parameters are represented at the bottom. Next in memory are 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.

- You can think of RAM as a stack of cafeteria trays, each representing a chunk of memory. Whenever a function is called, another tray is added to the top of the stack. As soon as that function returns, the tray is removed from the stack.
- This is a very brief look at what is known in computer science (quite aptly) as the *stack*. 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 along with what's called the text segment of your program—its representation in binary after being compiled. As we'll see soon, there also exists an area of memory called the *heap* from which you can explicitly borrow chunks within your program.
- Whereas the heap grows downward, the stack grows upward. If either grows too much, the two will collide with each other and your program will most likely fail, probably with a *segmentation fault* which indicates that two segments of memory have overrun each other. Often a file named `core` containing the contents of memory at the time of the segmentation fault will also be dumped.
- Incidentally, foo is a nonsense word that computer scientists use as a placeholder. There's also bar, baz, qux, and more.

4 Command-line Arguments (60:00–74:00)

- You may have noticed in some of our programs that we defined `main` as taking two arguments instead of `void`:

  ```
  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.
- Incidentally, argv stands for argument vector and argc stands for argument count.

4.1 `argv1.c`

- `argv1.c` makes use of `main`'s inputs `argv` and `argc`:

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

`argc` holds the number of command-line arguments that you passed to the program. By default, the 0^{th} argument is always the name of your program, so `argc` will always be at least 1.

- Recall that we've been using `string` as a shorthand for `char *`. A `char *` is actually a *pointer* which we'll discuss more in depth next week.
- The square brackets after `argv` indicate that it's a special type of variable that has multiple values inside of it. In order to access those values, you can index into it by specifying a number between the square brackets. `argv[0]`, for example, gives the 0^{th} argument which is always the name of the program.
- `argv` is an array, equivalent to an inventory in Scratch. Arrays are 0-indexed, so if `argv` has `n` elements, then `argv[n-1]` is the last element. Trying to access `argv[n]` will cause problems and may even leave your system vulnerable to being compromised. The Wii hack we mentioned last time used this exploit which is called a buffer overrun attack.
- Now, looking at `argv1.c`, we can see that it prints the command-line arguments that were passed to it.
- If we change the loop's terminating condition to `i <= argc`, we'll be indexing off the end of the array. Compiling and running the program will

cause a segmentation fault because you are touching memory that doesn't belong to you.

4.2 argv2.c

- `argv2.c` demonstrates what a string actually is:

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

Notice that in the first part of parentheses after the second `for`, we initialize not just `j`, but also `n`. We do this simply by separating them with a comma. `strlen` returns the length of a string. Because `argv` is actually an array of strings, we know that `argv[i]` is a string. It turns out that using bracket notation, you can also index into a string and return one of its characters.

- 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.

4.3 Cryptography

- 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. Caesar's cipher was popularized in A Christmas Story with Ralphie's decoder ring. Essentially, if you rotate each letter of the alphabet by a certain number called a key, you can write encrypted text that the recipient can decode if he knows the key.
- Of course, you can crack Caesar's cipher simply by brute force trying all the possible keys of which there are only 26.
- During World War II, the enigma machine implemented cryptography mechanically. The secret to the Germans' code lay in the machine itself, so the Allies endeavored to steal one from a U-boat.
- But what about Facebook or online banking? How can we encrypt our messages to them without agreeing on a key beforehand? This is the secret of public-key cryptography which we'll delve more into next week.