Computer Science 50
Fall 2009
Scribe Notes

Week 1 Friday: September 11, 2009
Andrew Sellergren

**Contents**

Computer Science 50
Fall 2009
Scribe Notes

Week 1 Friday: September 11, 2009
Andrew Sellergren

## 1 Announcements (0:00–7:00)

- This is CS 50.
- 0 new handouts.
- This is our last Friday class! For the rest of the semester, we'll meet only on Mondays and Wednesdays.
- Office hours for Problem Set 1 will begin this Sunday and will be held Sunday through Thursday nights. Check the course website to find out how many TFs will be on staff for a particular night.
- Supersections on Monday, Tuesday, and Wednesday. For this first week, we'll simply be holding large sections which anyone is welcome to attend. For the weeks to follow, you'll be attending your assigned section.
- Starting this Sunday, we'll also hold the first of our code walkthroughs. These are meant to help you get started with the standard editions of the problem sets so as to avoid the question "Where do I begin?" at office hours.[1] These will also be filmed and put online.
- Problem Set 1! You'll find that we hold your hand[2] as you're getting bootstrapped with Linux and C. This won't always be the case, so enjoy the long PDF while you have it! This problem set consists of several challenges:
  - Implement a program to validate ISBN numbers.
  - Implement a program to determine how a cashier should make change with the minimum number of coins possible.
  - Implement Mario's pyramid![3]
- Shuttleboy's SMS feature has been revived! To use it, send a text message to 41411 beginning with SBOY and followed by something like MEM and QUA (the first three letters of your origin and destination stops). Some fancy phone-switching equipment will route the message to our servers where the origin and destinations will be parsed and the schedule lookup is performed.

## 2 Back to C (7:00–34:00)

- We'll pick up where we left off in writing C programs. Recall that we learned how to SSH (using Terminal on a Mac or PuTTY on a PC) which took us to the command prompt on the NICE servers. From there we

[1] "Begin at the beginning and go on till you come to the end: then stop."
[2] Well, all of the staff but me will hold your hand. Sorry, I'm not a big fan of sweaty palms.
[3] Check out Super Mario Bros. in 5 minutes!

opened up what David calls a "tiny little" program named Nano.[4] At the top of the program we had some comments and preprocessor directives along with our declaration of the main method. And we ran GCC to compile our code into an executable program.

- Let's take a look again at the `man` page entry for `printf`:

  ```
  int printf(const char *format, ...);
  ```

  So `const` just means constant—don't worry about it for now. `char` stands for character. When we want multiple characters such as a word, a sentence, a paragraph—more generally, a *string*—we'll use the `char *` type. The ... stands for the space where we'll put the optional arguments which are substituted in for our placeholders. Recall our short list of escape sequences from last time:

  - `\n`
  - `\r`
  - `\t`
  - `\''`
  - `\\`

- We went on to discuss data types. In Scratch, data types didn't really exist. Whether we wanted to store a number or a string, we didn't have to specify how it would be stored. Not so with C, however. This *strict data typing* should be considered an advantage of C. It prevents you from performing illegal operations on variables (e.g. arithmetic on a string) and allows the compiler to optimize your program by making decisions about where in memory your variables will be stored. Below are a few of the built-in data types we have discussed:

  - `char` *A single character.*
  - `double` *A 64-bit real value (something with a decimal point).*
  - `float` *A 32-bit real value (with a decimal point).*
  - `int` *A 32-bit integer.*

    † `long, short, signed, unsigned`

  If you ever hear references to 32-bit or 64-bit architectures, know that it has to do with the size of integers on those systems. More specifically, it relates to the size of registers, small pieces of memory in the CPU. `long` is the same as `int`, so if you want 64 bits, you need a `long long`. `short` goes the other direction—it's only 16 bits. `signed` and `unsigned`, respectively,

[4] Don't blame David. He's just doing his part not to deplete the planet's rapidly shrinking resource of funny. Thanks for protecting our natural resources, David.

refer to integers that can store negative values or not, respectively. An `unsigned` can store numbers from 0 to 4 billion whereas a `signed` can stored numbers from −2 billion to 2 billion.

- Here's a quick glance at some of the format strings which you can provide to `printf`:
  - `%c`
  - `%d`
  - `%e`
  - `%E`
  - `%f`
  - `%s`
  - `%u`
  - `%x`

  Generally you'll only be using `%c` for `char`, `%d` for `int`, and `%s` for `char *`.

- All the usual arithmetic operators are available to you:
  - `+`
  - `-`
  - `*`
  - `/`
  - `%`

  The `%` stands for modulus, which returns the remainder after division. This is actually an incredibly useful trick, you might be surprised to find out!

- Operators in computer science have a property called *precedence*. This just refers to the order in which they are performed. When in doubt, you can always enclose operations in parentheses to guarantee that they will be executed before others. Don't worry about memorizing the table from the lecture notes, although it might be a useful thing to include on the cheat sheet you get for exams! [5]

- You might think that arithmetic operators are pretty straightforward, but take a look at `math3.c` to find out why you're DEAD WRONG:

[5] Or, if you prefer, you can come up with your own mnemonic just like Please Excuse My Dear Aunt Sally. Here's mine: `++sizeof<<!=`. Catchy, huh?

```
/**************************************************************************
* math3.c
*
* Computer Science 50
* David J. Malan
*
* Computes and prints a floating-point total.
*
* Demonstrates loss of precision.
**************************************************************************/

#include <stdio.h>

int
main(int argc, char *argv[])
{
    float answer = 17 / 13;
    printf("%.2f\n", answer);
}
```

If we compile and run this program we get 1.00 as output. What went wrong? Well, the way we've written it, the numbers 17 and 13 are being stored as integers. If we instead write 17.0 and/or 13.0, then the compiler will store them as floating-point values. We see this in `math4.c`. What about the `.2`? With this syntax, we're specifying the number of digits after the decimal point that will be displayed. What if we wrote `.98` instead? Well, 98 digits will be printed out, but most of the trailing ones will be zeroes. The size of a `float` is the limiting factor! Don't worry about exactly how limiting, just realize that the limit exists and that rounding errors are common in C (and every programming language for that matter). Don't compare floating-point values against each other if exactness is paramount!

- Instead of writing 17.0 and 13.0, we can also explicitly *cast* the numbers 17 and 13 as `float`'s. We do this with the following syntax, seen in `math5.c`:

```
/**************************************************************************
* math5.c
*
* Computer Science 50
* David J. Malan
*
* Computes and prints a floating-point total.
*
* Demonstrates use of casting.
**************************************************************************/
```

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    float answer = 17 / (float)13;
    printf("%.2f\n", answer);
}
```

Casting will become useful when we want to convert between alphabetic characters and numbers, as with ASCII. You'll be introduced to this for Problem Set 2.

- Among many others, there are some data types missing from C:
  - `bool`
  - `string`

  How might we simulate a boolean variable in C? Well we could use only the first bit of an `int` to store either 0 or 1. But we'd be wasting 31 bits in doing so. We could use a `char`, which is only 8 bits. In fact, we've done just that in CS 50's library. Underneath the hood, a `bool` is a `char` and a `string` is actually a `char *`. We've also made things easier on you by implementing some very useful functions:
  - `char GetChar();`
  - `double GetDouble();`
  - `float GetFloat();`
  - `int GetInt();`
  - `long long GetLongLong();`
  - `string GetString();`
- Let's reexamine the use of `GetString()` in `hai3.c`:

```
/**************************************************************************
 * hai3.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Says hello to whomever.
 *
 * Demonstrates use of CS 50's library and standard input.
 **************************************************************************/
```

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

int
main(int argc, char *argv[])
{
    printf("State your name: ");
    string name = GetString();
    printf("O hai, %s!\n", name);
}
```

Notice that, unlike `printf()`, `GetString()` doesn't take any input, but it does produce output. `GetInt()` is used in a similar fashion in `adder.c`:

```
/**************************************************************************
 * adder.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Adds two numbers.
 *
 * Demonstrates use of CS 50s library.
 **************************************************************************/

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

 int
 main(int argc, char *argv[])
 {
     // ask user for input
     printf("Give me an integer: ");
     int x = GetInt();
     printf("Give me another integer: ");
     int y = GetInt();

     // do the math
     printf("The sum of %d and %d is %d!\n", x, y, x + y);
 }
```

Don't forget that when we compile this, we need to include the `-lcs50` flag or we'll get the "undefined reference" error. Once we've done this, we see that the program takes two inputs and then displays their sum. Notice that we can embed the arithmetic expression `x + y` as an argument to the `printf` function.

- Now we'll challenge you to implement a program that converts a user's temperature input from Fahrenheit to Celsius. Use the `GetFloat()` function and the equation $°C = (5/9) \times (°F - 32)$. Hopefully, after a few minutes, your program looks something like the following:

```
1: /**************************************************************************
2: * f2c.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Converts Fahrenheit to Celsius.
8: *
9: * Demonstrates arithmetic.
10: **************************************************************************/
11:
12:
13: #include <cs50.h>
14: #include <stdio.h>
15:
16: int
17: main(int argc, char *argv[])
18: {
19:     // ask user user for temperature in Fahrenheit
20:     printf("Temperature in F: ");
21:     float f = GetFloat();
22:
23:     // convert F to C
24:     float c = 5 / 9.0 * (f - 32);
25:
26:     // display result
27:     printf("%.1f F = %.1f C\n", f, c);
28: }
```

  Notice that we write 9.0 because if we do integer division with 5 and 9, we'll always get 0 (the decimal part is rounded down). We write `%.1f` to limit the values to one decimal place each.

## 3 Conditions, Booleans, and Loops (34:00–72:00)

- One quick sidenote: the use of `//` denotes a comment just as `/*` and `*/` do, but only for a single line of text. Again, we can't stress enough the importance of commenting your code! It's useful to both us TFs and you programmers—you're not going to remember how it all works when you come back to it years later.
- Let's take a look at the use of conditions in `conditions1.c`:

```
1: /**************************************************************************
2: * conditions1.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Tells user if his or her input is positive or negative (somewhat
8: * innacurately).
9: *
10: * Demonstrates use of if-else construct.
11: **************************************************************************/
12:
13: #include <cs50.h>
14: #include <stdio.h>
15:
16: int
17: main(int argc, char *argv[])
18: {
19:    // ask user for an integer
20:     printf("Id like an integer please: ");
21:     int n = GetInt();
22:
23:     // analyze users input (somewhat inaccurately)
24:     if (n > 0)
25:         printf("You picked a positive number!\n");
26:     else
27:         printf("You picked a negative number!\n");
28: }
```

Notice we can omit the curly braces around our conditional statements so long as they don't exceed one line each. Indenting isn't enough! If you added another `printf` line after the first one in the `else` block, it will **always** execute, even if the number is negative.

- Can you spot the bug in `conditions1.c`? Looks like we're not properly handling the case where the user provides the number 0. After all, it's neither positive nor negative. We can fix this by using an `else if` block as well:

```
1: /**************************************************************************
2: * conditions2.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Tells user if his or her input is positive or negative.
```

```
 8: *
 9: * Demonstrates use of if-else if-else construct.
10: ****************************************************************************/
11:
12: #include <cs50.h>
13: #include <stdio.h>
14:
15: int
16: main(int argc, char *argv[])
17: {
18:     // ask user for an integer
19:     printf("Id like an integer please: ");
20:     int n = GetInt();
21:
22:     // analyze users input
23:     if (n > 0)
24:         printf("You picked a positive number!\n");
25:     else if (n == 0)
26:         printf("You picked zero!\n");
27:     else
28:         printf("You picked a negative number!\n");
29: }
```

As you can see, this is a bug that's very easily fixed, but it's not necessarily one you would've spotted right away, especially as you were writing the code. That's why we encourage you bang on your code—to really test it before submitting to make sure you can't break it with inputs you weren't expecting the user to provide.

- Here, the logic dictates that we have only three cases. Generally it's a good rule of thumb to not string too many `else if` blocks back to back. We can, however, handle more than three cases, as we do in `nonswitch.c` by using a few boolean operators, as well:

```
 1: /****************************************************************************
2: * nonswitch.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Assesses the size of users input.
8: *
9: * Demonstrates use of Boolean ANDing.
10: ****************************************************************************/
11:
12: #include <cs50.h>
```

```
13: #include <stdio.h>
14:
15: int
16: main(int argc, char *argv[])
17: {
18:     // ask user for an integer
19:     printf("Give me an integer between 1 and 10: ");
20:     int n = GetInt();
21:
22:     // judge users input
23:     if (n >= 1 && n <= 3)
24:         printf("You picked a small number.\n");
25:     else if (n >= 4 && n <= 6)
26:         printf("You picked a medium number.\n");
27:     else if (n >= 7 && n <= 10)
28:         printf("You picked a big number.\n");
29:     else
30:         printf("You picked an invalid number.\n");
31: }
```

Pretty straightforward. Certainly it works as it's supposed to, but is there a better way to do this from a style or readability standpoint? You betcha![6] We do so using a construct called a *switch*, as we see in `switch1.c`:

```
1: /**************************************************************************
2: * switch1.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Assesses the size of users input.
8: *
9: * Demonstrates use of a switch.
10: **************************************************************************/
11:
12: #include <cs50.h>
13: #include <stdio.h>
14:
15: int
16: main(int argc, char *argv[])
17: {
18:     // ask user for an integer
```

[6] And yes, there's always a better design. Your program will never quite be perfect. Le sigh.

```
19:     printf("Give me an integer between 1 and 10: ");
20:     int n = GetInt();
21:
22:     // judge users input
23:     switch (n)
24:     {
25:         case 1:
26:         case 2:
27:         case 3:
28:             printf("You picked a small number.\n");
29:             break;
30:
31:         case 4:
32:         case 5:
33:         case 6:
34:             printf("You picked a medium number.\n");
35:             break;
36:
37:         case 7:
38:         case 8:
39:         case 9:
40:         case 10:
41:             printf("You picked a big number.\n");
42:             break;
43:
44:         default:
45:             printf("You picked an invalid number.\n");
46:     }
47: }
```

Functionally, this program is identical to `nonswitch.c`. Arguably, though, it's more readable, albeit longer. It's probably a toss-up between these two versions, but so long as your code is pret-printed and reasonably well-stylized, we won't be picky. Ultimately we'll post a style guide on the course website which should give some useful starting points.

- In the above program, each of the `case` statements is compared with the variable provided to `switch` at the beginning of the block. If the case matches the variable, then its lines of code are executed. Notice that the cases lump together unless we explicitly type `break`. Thus 1, 2, and 3 fall together, 4, 5, and 6 fall together, and 7, 8, 9, and 10. This is a common source of bugs in programs! Don't forget the `break` statements!
- `switch2.c` is similar to the above except that it switches on a `char` rather than an `int`.
- Now let's take a look at loops. The syntax for loops is exemplified below:

```
for (int i = 0; i < 200; i++) {...}
```

Before the first semicolon, we are initializing a variable which will be our iterator or counter. Between the two semicolons, we're providing a threshold which, once reached, will cause the loop to be terminated. Finally, we provide code to update our iterator. So each time the loop executes, the variable `i` will be incremented by 1 until it reaches 200. We can do some interesting things with this. We could print out asterisks or the actual value of our iterator. Realize that the last value of `i` will be 199 because it's the highest value which is still strictly **less than** 200.

- What if we change our update code to `i--`? Will it go forever in the same direction? At some point, we'll run out of bits and the value will flip from negative to positive. If we update exponentially rather than geometrically (`i = i * 2`) and set our threshold as `i < 0`, our program will actually come to an end quickly (once it exceeds the largest possible positive value that can be stored in an `int` and it flips to negative).
- One use of this might be a progress indicator, as we've implemented in `progress1.c`:

```
1: /**************************************************************************
2: * progress1.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Simulates a progress bar.
8: *
9: * Demonstrates sleep.
10: **************************************************************************/
11:
12: #include <stdio.h>
13: #include <unistd.h>
14:
15: int
16: main(int argc, char *argv[])
17: {
18:     // simulate progress from 0% to 100%
19:     for (int i = 0; i <= 100; i++)
20:     {
21:         printf("Percent complete: %d%%\n", i);
22:         sleep(1);
23:     }
24:     printf("\n");
25: }
```

- The `%%` is the escape syntax to print a literal percent character. `sleep()` is a function which simply pauses execution of the program for a given number of seconds. More interesting than this, though, is `progress2.c` which actually implements a form of animation:

```
1: /**************************************************************************
2: * progress2.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Simulates a better progress bar.
8: *
9: * Demonstrates \r, fflush, and sleep.
10: **************************************************************************/
11:
12: #include <stdio.h>
13: #include <unistd.h>
14:
15: int
16: main(int argc, char *argv[])
17: {
18:     // simulate progress from 0% to 100%
19:     for (int i = 0; i <= 100; i++)
20:     {
21:         printf("\rPercent complete: %d%%", i);
22:         fflush(stdout);
23:         sleep(1);
24:     }
25:     printf("\n");
26: }
```

  By using a carriage return (`\r`) instead of the newline character (`\n`), the next value overwrites the previous value on our screen. We could do this to animate the severe loss Harvard suffered to its endowment. Note that we'll have to use a `long long` to store the value of the endowment since it's larger than the 4 billion max of an `int`.

- One thing we failed to mention earlier was the use of the `fflush` function. Even when we give the computer an explicit instruction, it might actually put it off for the time being in order to save resources. This is often the case with the `printf` function, which won't immediately update standard out. Instead, it will store values in a *buffer* until the buffer is full and only then will it spit out the value. The call to `fflush` shortcuts this and forces standard out to be updated immediately. Also, to use the `sleep` function, we'll need to include the `unistd.h` library. We know which library to

include by checking its `man` page. Just type `man` and then the name of the function at the command line to find out more about a function.

- Two other types of loops are `while` and `do while`. `while` takes a simple terminating condition as its argument. When you use it, you must be careful to include the update code within the `while` block and initialize the iterator before the block.
- `do while` has a particular use. The `while` block comes after the `do` block, and, as you might expect, executes after it as well. This is useful when we want to guarantee that some block of code be executed **at least** once no matter what. One circumstance in which this comes in handy is for taking user input. Take `positive1.c` for example:

```
1: /**************************************************************************
2: * positive1.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Demands that user provide a positive number.
8: *
9: * Demonstrates use of do-while.
10: **************************************************************************/
11:
12: #include <cs50.h>
13: #include <stdio.h>
14:
15: int
16: main(int argc, char *argv[])
17: {
18:     int n;
19:
20:     // loop until user provides a positive integer
21:     do
22:     {
23:         printf("I demand that you give me a positive integer: ");
24:         n = GetInt();
25:     }
26:    while (n < 1);
27:    printf("Thanks for the %d!\n", n);
28: }
```

Obviously, we want to want to ask the user for his input at least once no matter what. Now we'll either continue to execute the loop (which will ask the user for input again) if the user didn't provide the input we were looking for (in this case a positive integer). Notice if we put in completely

invalid input, though, like a string, we'll get a different reprompt. This is because of the error-checking we have built in to `GetInt()`. Don't rely on this though!

- `positive2.c` implements the exact same program but with the use of a boolean variable:

```
1: /**************************************************************************
2: * positive2.c
3: *
4: * Computer Science 50
5: * David J. Malan
6: *
7: * Demands that user provide a positive number.
8: *
9: * Demonstrates use of bool.
10: **************************************************************************/
11:
12: #include <cs50.h>
13: #include <stdio.h>
14:
15: int
16: main(int argc, char *argv[])
17: {
18:     bool thankful = false;
19:
20:     // loop until user provides a positive integer
21:     do
22:     {
23:         printf("I demand that you give me a positive integer: ");
24:         if (GetInt() > 0)
25:             thankful = true;
26:     }
27:     while (thankful == false);
28:     printf("Thanks for the positive integer!\n");
29: }
```

Be careful. If we were to write `thankful = false` as our `while` condition, then we're not **comparing** the value of `thankful` to the value `false`, we're actually **assigning** the value of `false` to `thankful`. So no matter what input we provide, it will always thank us for the positive integer.

- `positive3.c` is a final example that demonstrates the use of the `!` or bang operator. This inverts the value of whatever expression comes after it. So if we write `while (!thankful)`, it reads as "while not thankful," which actually makes for pretty readable code.

- If you haven't already, be sure to play around with Google Earth. How is this relevant to CS 50? Toward the end of the semester, you'll be asked to implement a "mashup." You'll be required to use the *Application Program Interfaces* (APIs) provided by Google. This is their library of code along with documentation which you can plug in to your own programs. An example of a mashup is one which David threw together which took all of the home addresses of those who have submitted Problem Set 0 so far in a *comma-separated values* (CSV) file, converts it to a KML file using a PHP script (which is surprisingly readable now that you know some C) and plugs it in to Google Maps. Now we can see you!