Computer Science 50
Fall 2009
Scribe Notes

Week 1 Wednesday: September 9, 2009
Andrew Sellergren

Contents

1 Announcements (0:00–5:00)

- This is CS 50.
- 1 handout. Ignore the line numbers next to the code–they're just for reference.
- Office Hours this Wednesday and Thursday to talk about the course or your Scratch projects.
- 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. To get your section assignment, follow these instructions. Note that we offer three tracks of sections, one for "those less comfortable," one for "those more comfortable," and one for those in between. The FAS Sectioning tool will show some 30+ sections and each will be designated as one of these three categories according to its comments field. Realize that you can't go wrong with the "in between" section.
- Sensor boards. These are only for the Hacker Edition, so don't feel pressured to try them out.
- The data we gather from the survey for Problem Set 0 is really just for fun. We want to visualize that data in several different ways, for example, a Google Maps mashup of birth cities. Of the 30 people who have submitted the problem set so far, most do not own an iPhone, most are running Windows, and none of the Mac users have upgraded to Snow Leopard. It's a good thing that most do not have an iPhone, because it makes 617-BUG-CS50 worthwhile! Coming soon also will be wake-up calls and dining hall menus.

2 Scratch Meets C (5:00–20:00)

- Unlike Java and scripting languages like PHP, C is very low-level, giving you fine-grained control over where you put things in memory. It's just one level removed from assembly language, which is just one level removed from binary! In addition to affording you great control, C offers you great opportunity to screw up. Most of the exploits and hacks, even today, are the result of mistakes that programmers made in C or C++.
- Recall our first C program from last week. It's a good example of a statement in C:

```
#include <stdio.h>

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

```
{
    printf("hello, world!\n");
}
```

- All of the same building blocks in Scratch—statements, loops, boolean expressions, conditions—exist in C as well. The only real difference is in how you interact with the program you're making. Many of the errors you'll make in C are simply syntax errors if you forget a semicolon somewhere, for example.

- Boolean expressions are much the same in C as in Scratch. The less-than (<) and greater-than (>) operators are the same. One difference is that the "and" operator is represented as `&&` in C.

- Conditions in Scratch, which were represented by interlocking puzzle pieces, are encapsulated by curly braces in C. Instead of nesting if–else blocks, you'll be writing if–else-if–then blocks.

- Instead of the forever loop in Scratch, we have the syntax `while(1)` in C. What does this mean exactly? Well, the code inside the `while` block will repeatedly execute as long as its condition is true and, since 1 is synonymous with true, it will execute indefinitely. Hence, forever. Generally infinite loops are not desirable.

- The repeat block in Scratch can be represented by a `for` loop in C like so:

  ```
  for (int i = 0; i < 10; i++)
  {
      printf("O hai!\n");
  }
  ```

  These lines of code simply cause "O hai!" to be printed 10 times. The `for` loop continues to execute as long as `i` is less than 10 and since `i` is incremented by 1 every time the loop executes (`i++`), after 10 executions, `i` will no longer be less than 10.

- Variables in C are fairly straightforward:

  ```
  int counter = 0;
  ```

  Here we're declaring what type of variable we want (`int`), the name of the variable (`counter`), and its default value (`0`).

- Why is it called `printf` rather than just `print`? The f stands for formatted because `printf` is used to print formatted strings. For example:

  ```
  printf("%d\n", counter);
  ```

The %d tells printf that what we want to display is a numerical value (d for decimal).

- Recall that we used a variable called "inventory" in Scratch to store a series of related variables—fruits in the case of FruitcraftRPG. This inventory can be implemented as an array in C:

  ```
  char *inventory[SIZE];
  inventory[i] = "Orange";
  ```

  In C, because we're so close to the hardware, we have to tell the compiler how big we want our array to be when we first declare it, hence the reference to SIZE, which is actually a constant that will represent the number of bits. When we want to store something in the array, we access it using bracket notation. The second line above stores the string "Orange" at the i^{th} location in the array. char * by the way, is the computer scientist name for string, or sequence of letters.

- A big gotcha: in programming, = doesn't generally mean equality, but rather assignment. That is, it means take the thing on the right and put it in the thing on the left. So programmers often say "gets" rather than "equals" when reading =.

3 NICE and Teh Cloud (20:00–41:00)

- There was a time when programming meant writing the actual 0's and 1's on punchcards rather than plain English to be fed into a compiler. In those days, you would gather your punchcards together, take them to the Science Center where they would be fed into the mainframe computer, and leave them overnight to run. If your program was buggy, you would only find out the next morning when the results of your program were just one long debugging report.[1]

- At the end of the day, the computer still only understands 0's and 1's. This is why we have the process of compiling, which turns your programming syntax into binary. We did this last week when we ran the gcc command.

- What's neat about Macs is that they have a built-in terminal because they're based on Unix, which is essentially a command-line interface (CLI). So if you have a Mac, you can simply open up the Terminal application. If you're running Windows, open up your SSH client—PuTTY, for example.

- Instead of attempting to normalize all of your personal computers, we've provided a standardized development environment on Harvard's servers.

[1] My mom continues to remind me to this day that she was one of the original programmers who wrote in punchcards. She usually reminds me of this when I'm showing her how to use iTunes.

This is NICE. No, really, it's NICE, i.e. New Instructional Computing Environment. It's been "new" for about 10 years now, so don't ooh and ahh just yet.

- We begin our work on NICE not just for historical reasons, but also because at semester's end, you'll have a place to host your programs, at least until you graduate. Toward the middle of the semester, however, we'll be migrating from NICE to CS 50's cloud, which is a fancy term for the cluster of servers that we ourselves have set up.[2] CS 50's cloud consists of a few Dell servers with 32 GB of RAM and 6 TB of storage space, along with an Apple XServe. We'll give you your own accounts on `cloud.cs50.net`.

- Why do we call them clusters? Well, with external IP addresses and DNS hostnames like `nice.fas.harvard.edu` and `cloud.cs50.net`, we give the illusion that you're connecting to just one computer each time. That's not the case, however. You may be connecting to any one of several computers, each of which has access to your files.

- So to connect to NICE or Teh Cloud,[3] you'll open up Terminal or PuTTY and type `ssh username@nice.fas.harvard.edu`. This tells your computer to open an encrypted connection to the server.

- Not to worry, all these commands will be rehashed in the PDF for Problem Set 1. The command `ls`, for example, lists the content of the folder that you're currently in. `mkdir` makes a directory, `cd` changes the directory you're in. To write a program, you'll open a text editor—`nano` is the one we'll recommend if you're new to all this. So type `nano hello.c` for example.

- If we now type out the lines of code we introduced last week and save the file, we can compile it by typing `gcc hello.c`. This will create a file called `a.out` which will have an asterisk next to it, indicating that it's executable. You can open `a.out` in a text editor, but you'll just see a bunch of gobbledygook, otherwise known as binary.[4] Be careful if you open a binary file like this, because any ASCII characters you insert will likely break your program.

- A short list of Linux commands:
  - `cd` *Change directory.*
  - `cp` *Copy.*

[2]It was mostly me. David doesn't really know what he's doing, to be honest. He's mostly a figurehead. Actually, I usually write these scribe notes *before* class and he just follows along. Oops, I think I've said too much.

[3]Yes, that's twice now. It's "Teh," not "The." Psth, n00b.

[4]Allow me to put the rumors to rest. Yes, I can read straight binary. Remember Tank from The Matrix? He's loosely based on me. I say "loosely" because I would never allow a punk traitor like Cypher to get the best of me.

  - `ls` *List files.*
  - `mkdir` *Make directory.*
  - `mv` *Move/rename file.*
  - `pwd` *Present working directory (your current location)*
  - `rm` *Remove/delete file.*

- Note if we run `gcc` with the option flag `-o` followed by a new name for our program, perhaps `hello`, the output of the compiler will be named `hello` rather than `a.out`.
- Another useful shortcut on Linux is to begin typing a command and hit TAB which will auto-complete your command or show you a list of possible commands.
- Good design prescribes that we place our code in multiple related files rather than one large file we have to scroll through. But how do we organize these files appropriately? And how does the compiler know which files to include when it's compiling? This where the Makefile comes in, which are instructions provided to the `make` command. This accomplishes the same compiling as the `gcc` command, but with more structure and any additional instructions we choose to provide.
- A short list of text editors:
  - Nano
  - Vim
  - Emacs

4 More Code! (41:00–90:00)

- Let's say we want to play with this week's source code in our home directory. We'll run the following command:

  ```
  cp -r ~cs50/pub/src/lectures/1/ .
  ```

  The tilde (~) followed by `cs50` designates CS 50's home directory. Then, we know the directory structure (because we've told it to you). And inside the `lectures` directory, there are directories for each of the weeks of the course. So we're going to copy all the files from the week 1 directory to our current directory, which we designate with a single period. The `-r` flag stands for *recursive*, which tells the command to copy subdirectories as well.
- Let's take a look at the pre-prepared `hai1.c`:

```
/****************************************************************************
 * hai1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Says hello to the world.
 *
 * Demonstrates use of printf.
 ***************************************************************************/

#include <stdio.h>

int
main(int argc, char *argv[])
{
    printf("O hai, world!\n");
}
```

All of the junk at the top between the `/*` and the `*/` are what are called *comments*. They're ignored by the compiler, but are very useful to programmers who are reading your source code (including yourself). We will insist that all of your programs be well-commented!

- Note that our code is indented in key places. Everything which is encapsulated in the `if` condition is indented to indicate so. The computer most likely doesn't care about this indentation but it makes the code easier for us to read.

- We will also insist that your code be pretty-printed. That means, for one, that your lines of code never extend beyond 80 characters. That way, it will display properly on most all screens.

- One feature of the text editors we've mentioned, as opposed to say, Notepad, is syntax highlighting. By default, Nano and Vim will make assumptions about the roles of certain words you've written, and highlight variable types in one color, comments in other, etc. It makes for easier reading.

- So what exactly is going on with the first real line of code in our program above? It's called a *pre-processing directive*. What we're asking the compiler to do is to go look for a library of code called `stdio.h` and compile it with our program. This library includes the definition of our `printf` function.

- Why is the file extension `.h` instead of `.c`? The file we're asking to include is actually a *header* file which simply contains a summary of information about the functions defined in the actual file. This is easier for a human to digest. The actual function definitions are most likely in `stdio.c`.

- Programs are comprised of *functions*, which you can think of as miniature programs. Functions in C are the equivalent of scripts in Scratch. And just as the green flag is the default script in Scratch, so is `main` the default function in C. What about all the other junk along with `main`? We'll gloss over this for now. Just consider it a copy-paste job for the moment.

- Let's take a look at `hai2.c`:

```
/****************************************************************************
 * hai2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Says hello to just David.
 *
 * Demonstrates use of CS 50's library.
 ***************************************************************************/

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

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

  What's the difference between this and `hai1.c`? Hopefully you can figure out that this one says hi only to David. How do we do this? We're simply assigning the value of "David" to a variable of type `string`. Be careful: in C, you must encapsulate strings in double quotes, not single quotes (in HTML, it doesn't matter).

- A little more interesting is the third version, `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);
}
```

Here, we invoke a function called `GetString()`, which, as its name implies, gets a string from the user. So our functions not only can execute statements, they can *return* their output as well. What we're doing is asking to store the return value of `GetString()` in our variable called `name`. Then we'll print out `name` as before.

- When we try to compile `hai3.c` with the command `gcc`, we get an error. Notice that we're asking not only for the `stdio.h` library, but also the `cs50.h` library. This is where the definition of `GetString()` resides. But how does the compiler know where to look for this? We need to tell it by adding the flag `-lcs50` when we run `gcc`. This means to *link* the CS 50 library when we're compiling.

- What do you do if you don't know how a function works? RTFM![5] Specifically, check out the one on our reference page. The page for `printf`, for example, contains all the formatting codes for inputs (e.g. `%f`, `%c`, etc). The pages also include sample code and function signatures, which are tremendously useful.

- Take note of these escape sequences:
  - `\n`
  - `\r`
  - `\t`
  - `\''`
  - `\\`

  The first two are caught in the war between Linux and Windows. The first is a newline return, which is just like hitting Enter. The second is a carriage return which is like hitting Enter and then repositioning the cursor all the way to the left on the screen. The third is a tab, the fourth quotation marks, the fifth is how you place a literal backslash.

[5]The F stands for frabjous.

- Let's try doing some math in `math1.c`:

  ```
  /****************************************************************************
   * math1.c
   *
   * Computer Science 50
   * David J. Malan
   *
   * Computes a total but does nothing with it.
   *
   * Demonstrates use of variables.
   ***************************************************************************/

  #include <stdio.h>

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

  If we compile and run this program, nothing happens. Is it working properly? In fact, it is, but it gives no output since we haven't told it to. Also, the compiler will yell at us for not doing anything with the variable `z`. Our next version, `math2.c` will actually print out the result of our computation.

- Let's take a look at some different variable types with `math3.c`:

  ```
  /****************************************************************************
   * sizeof.c
   *
   * Computer Science 50
   * David J. Malan
   *
   * Reports the sizes of C's data types.
   *
   * Demonstrates use of sizeof.
   ***************************************************************************/

  #include <stdio.h>

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

```
{
    // some sample variables
    char c;
    double d;
    float f;
    int i;

    // report the sizes of variables' types
    printf("char: %d\n", sizeof(c));
    printf("double: %d\n", sizeof(d));
    printf("float: %d\n", sizeof(f));
    printf("int: %d\n", sizeof(i));
}
```

The function `sizeof` gives us the number of bytes required to store each of these variable types. For historical reasons, a `long` is 4 bytes, the same as an `int`. If you want to store a 64-bit number, you'll need a `long long` or a `double`. How big a number can an `int` store? If it's `unsigned`, meaning it can't store negatives, it can store up to the number 4 billion. A `float`, although 4 bytes, can't store up to 4 billion because it uses some bits to store the numbers after the decimal place.

- Now, for some fun. There's a concept in programming (and frankly, in life) called *obfuscation*. With clientside programming languages like JavaScript, obfuscation is a technique of making your code purposely confusing so that it's hard to steal. There are even some obfuscation contests out there for C. To finish, we'll give you an example of an obfuscated program:

```
// http://www.ioccc.org/years.html

#include "stdio.h"
#define    e 3
#define    g (e/e)
#define    h ((g+e)/2)
#define    f (e-g-h)
#define    j (e*e-g)
#define k (j-h)
#define    l(x) tab2[x]/h
#define    m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
```

```
        if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
        else switch(m1-=h){
        case f:
            a=(b=(c=(d=g)<<g)<<g)<<g;
            return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
        case h:
            for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
        case g:
            if(n<h)return(g);
            if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
            else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
            if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
            return(o[b-g]%n+k-h);
        default:
            if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
            for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
        }
    }
```

Can you figure out what this program does? If you're dying to know, just compile it and find out. Or just watch the lecture video. Sheesh, do I have to do everything for you?