Computer Science 50
Fall 2010
Scribe Notes

Week 1 Wednesday: September 8, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 1 Wednesday: September 8, 2010
Andrew Sellergren

1 Announcements (0:00–1:00, 6:00–9:00)

- This is CS50.
- 1 new handout outside.
- We were mentioned on FlyBy yesterday as a great course to shop! As they mentioned, you're welcome to take CS50 pass/fail, you just need to have David sign your white study card or your pink slip before the 5-week deadline.
- Office hours are already in progress! Right now, they're located in the dining halls, but they'll be moving to other locations soon. Check out the schedule.
- We haven't yet assigned you to specific sections, but supersections will be happening on Sunday, Monday, and Tuesday of this coming week. Supersections are essentially the same as sections except they're open to the entire class, so their attendance might be greater than that of sections.

2 Small Bites (1:00–6:00)

- Let's begin with a cookie love story! Realize that even if this Scratch project, cutesy as it may be, appears overwhelming to you, you can bite it off one small piece at a time.[1]
- Before we use any puzzles pieces, the first step would be to find the graphic of a cookie sheet and place it in the correct position.
- Next, focus on a single cookie. How can we get him to dance? Probably by using some kind of loop that affects his position and orientation. After a half hour or hour, maybe we have a single cookie dancing in the middle of a cookie sheet with no music playing.
- Once you create a single dancing cookie sprite, you can right click on it and duplicate it. This will save you the time of reimplementing the dancing logic and will allow you to simply tweak how the sprite dances.
- After twenty seconds or so, the background image changes and the gingerbread cookies appear to move back and forth in sync with the music while hearts flow from bottom to top on the screen. The synchronization with the music might be achieved by trial and error (i.e. using the "wait" puzzle piece with a specific number of seconds you figured out from timing the music yourself) and the seemingly random starting positions of the hearts might be achieved with the pseudorandom number generator we mentioned last time.

[1] Mmm...small cookie bites.

- Avoid the temptation to take the whole project on in one fell swoop. You'll inevitably introduce bugs that are hard to track down and find yourself with a frustrating program that doesn't work and isn't easy to fix.

3 From Scratch to C (9:00–20:00)

- Although a so-called lower-level programming language like C might seem daunting with its semicolons, parentheses, and other syntactic conventions, the logic underlying its programs is the same as that of Scratch.
- Recall our first C program from last week:

```
#include <stdio.h>

int
main()
{
    printf("O hai, world!\n");
}
```

  The blue "say" puzzle piece from Scratch has no become `printf` and the orange "when green flag clicked" puzzle piece has become `main()`.

- The "forever" loop from Scratch can be recreated with a `while (true)` block This syntax purposefully induces an infinite loop. Whatever is within the parentheses is the `while` condition. As long as that condition evaluates to "true," the code within the `while` loop executes. Since the keyword `true` is always "true," the code within the loop always executes. A `while` loop is denoted in C between curly braces like so:

```
while (true)
{
    printf("O hai!\n");
}
```

- The "repeat" loop from Scratch is equivalent to a `for` loop in C that looks like so:

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

  This syntax declares an integer named `i` (a convention used for variables that are only used for counting) which is set to 0 to begin with. `i < 10` implies that the code within the loop will execute as long as `i` is less than 10. Finally, on each iteration of the loop, the statement `i++` increments `i` by one. All in all, this code cause "O hai!" to be printed 10 times.

- A Scratch loop that increments a variable and announces its value looks like this:

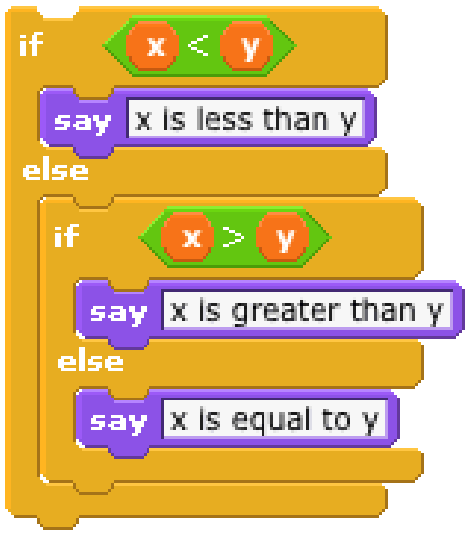
  In C, this loop would look like so:

```
int counter = 0;
while (true)
{
    printf("%d\n", counter);
    counter++;
}
```

  Here we declare a variable named `counter` and then create an infinite loop that prints its value then increments it.

- 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 C look much the same as they do in Scratch:



```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else
{
    printf("x is equal to y\n");
}
```

- 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:

```
string inventory[1];
inventory[0] = "Orange";
```

- Running a program written in C requires an extra step compared to Scratch. We must take our source code and pass it to a compiler which translates it to binary. This is the step that David actually failed to complete last week because the compiler (GCC) wasn't installed.[2]

4 Introduction to C (20:00–73:00)

4.1 Writing, Compiling, and Executing

- To write programs in C, we'll need the help of a text editor. CS50's text editor of choice is Nano. Nano is much like TextEdit on a Mac or Notepad on a PC except that it is a CLI (command-line interface) instead of a GUI (graphic user interface).
- To open a new C source code file named `hello.c` on a computer which has Nano installed, we run the command `nano hello.c` from the command line. When we do so, we'll see commands like "Get Help" and "Exit" listed at the bottom of the window. Each of these commands is preceded by a keyboard shortcut (e.g. `^G` and `^X`) written with a white background. The caret symbol (`^`) represents the Ctrl key. Note that "Write Out" means save.
- Once we write our very simple C program,

[2] Why, yes, David *does* have a PhD in computer science, why do you ask?

```
#include <stdio.h>


int
main()
{
    printf("O hai, world!\n");
}
```

we hit Ctrl + X to exit. We'll then be asked if we want to "save modified buffer" (we do) and we'll be prompted for a filename to save our text under. When we decide on one, we hit Enter and Nano will close.

- Now we're back at the command line. We need to pass our source code file to the compiler, which we do by running the command `gcc hello.c`. Although nothing appears to happen when we run this command, GCC has actually created a binary file named `a.out` that represents our executable program.
- To run this program that we've just created, we execute the command `./a.out`. The `./` tells our computer to look for a file `a.out` in the directory we're currently in, not in any other directory. When we run this command, we see the words "O hai, world!" printed out to the screen. Success!
- Now let's walk line by line through the source code of the program we just wrote. The `#include` line tells the compiler to bring someone else's code into our program so that we can make use of it. Notice that nowhere in our code do we explicitly tell the computer how to write words to the screen. This is because the `printf` function, which someone else wrote and is in a library called `stdio.h`, accomplishes this for us.
- When we write `int main()`, we're actually creating our own function named `main`. By default, this function is executed when our program is run. This is similar to anything that we placed under the "when green flag clicked" puzzle piece in Scratch.
- The curly braces denote where the `main` function begins and ends.
- To the `printf` function, we pass a few words that we want to be printed to the screen. We'll end these words with a `\n` character, which denotes a line break.
- So, to recap, the three commands we run from the command line in order to write a program with source code file `hello.c` are as follows:

  1. `nano hello.c`
  2. `gcc hello.c`
  3. `./a.out`

If you'd like to name your program `hello` instead of `a.out`, you would modify the above steps like so:

1. `nano hello.c`
2. `gcc -o hello hello.c`
3. `./hello`

The `-o` syntax is a so-called command-line switch or flag that toggles a program's options, in this case, the option to name the output file something other than `a.out`. Note that capitalization is important, as `-O` specifies a different option than `-o`, if any exists at all.

- As our programs get more complex, we'll have a need to specify multiple flags and pass numerous arguments to the compiler. To simplify this process, we'll actually use the command `make`, which will be a shortcut for all these flags and arguments via a separate file called a makefile.
- Up to this point, we've been writing and compiling programs on our local machine. That is to say, we haven't required an internet connection thus far (except to download the compiler software initially). However, it can be tedious to set up this programming environment on a local machine. Thus, it is common to set up a programming environment on a separate server that we can connect to via the internet. In Problem Set 1, you'll be setting up your own account on the course's server, known as the CS50 Cloud, where you'll be doing all of your programming this semester. At the end of the semester, if you choose to create a website for your final project, you'll have the option to host it on the CS50 Cloud. That is, all of the website's code will live on the cloud and you can register a different domain name (e.g. `isawyouharvard.com`) which will map back to our servers.

4.2 Some Jargon

- *Functions* are the term we'll use to denote miniature programs within our programs. Just like a mathematical function, these take one or more inputs and return some output. The first function we wrote was called `main`.
- The *standard library* is `stdio.h`, which we included in our first program in order to make use of the `printf` function.
- It turns out that taking input from the user is actually a somewhat complicated process in C. Because we feel that you should begin your exposure to C with more interesting tasks, we've written some functions to accomplish this for you. These are available to you in the CS50 library which lives at `cs50.h`:
  - `GetChar`

- `GetDouble`
- `GetFloat`
- `GetInt`
- `GetLongLong`
- `GetString`

These functions get input of different types (e.g. characters, doubles, floats, integers) from the user as he enters them on the keyboard.

4.3 Back to Coding

4.3.1 `hai2.c`

- To connect to the CS50 Cloud, we'll use a program called Terminal on a Mac or, alternatively, PuTTY or SecureCRT on Windows. More on this in Problem Set 1, but for now, know that we run a command `ssh malan@cloud.cs50.net`[3] to establish a secure connection to the course server. Once we've done so, we'll be presented with a prompt that looks like the following:

  ```
  malan@cloud(~):
  ```

  This prompt reminds us who we are (`malan`), what server we're connected to (`cloud`), and what folder we're currently in (`~` is shorthand for our home directory).

- Note that all of the source code for lectures, printed alphabetically by filename, is available both as handouts and on the course website.

- Instead of `nano`, David will be using `vim`, but they are both good text editors.[4]

- 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.
 **************************************************************************/
```

[3] Here and in all examples afterward, `malan` is interchangeable with your own CS50 username.

[4] Real programmers use Emacs.

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

int
main()
{
    string name = "David";
    printf("O hai, %s!\n", name);
}
```

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 this program includes not only the standard library by way of `stdio.h`, but also the CS50 library by way of `cs50.h`. 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 in `.c` files.
- In the first line of our `main` function, we declare a variable called `name` that is of type `string`. The `string` type holds a word or a chunk of text. To assign a value to a `string`, we enclose it in double quotes.
- When we call `printf` in this program, we write `%s` within the string that we wish to print out. Then, after the string, we have a comma followed by `name`. What we're doing here is passing more than one *argument* to `printf`.[5] The first argument is the string we wish to print out. Within that string, however, we've included a *formatting character*, namely `%s`. This tells `printf` to substitute something into our string. Here, the value of `name` will be inserted where the `%s` is. To insert multiple items into our string, we include multiple formatting characters in our string and pass the corresponding number of extra arguments to `printf`, in the order that we want them inserted. If the items we wish to insert in our string are of types other than `string`, we use different formatting characters. For example, `%d` for an `int`.

4.3.2 `hai3.c`

- By convention, the comment at the top of a source code file describes succinctly and in plain language what it is that a program does. You'll find this extremely useful when you come back to a program three months after writing it and can't remember what it actually does.

[5] An argument, recall, is simply an input to a function.

- Moving on to `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()
{
    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 can not only 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 like the following:

```
/tmp/cca0tzBk.o: In function ‘main’:
hai3.c:(.text+0x17): undefined reference to ‘GetString’
collect2: ld returned 1 exit status
```

  Although cryptic, this error seems to imply at least that the compiler can't find the definition of `GetString`. 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 is known as *linking* the CS 50 library when we're compiling. Why don't we need to write `-lstdio` when using the standard library? Because it's so common, `gcc` knows by default where to find this library.

- Now when we compile our program with the full command:

```
gcc -o hai3 hai3.c -lcs50
```

we get no errors. However, when we execute the command `hai3`, we get the following error:

```
-bash: hai3: command not found
```

Oops. The computer doesn't know where the `hai3` program lives. We need to tell it to look in our current directory by appending `./` to the beginning of the command. This is a testament to how stupid[6] computers really are. By default, your computer only knows to look for programs in a single or a few default directories. To run other programs which you've recently created, you need to tell the computer where to look.

- When we run `./hai3`, finally we see the text"State your name:" and can enter a name like Cansu.[7] When we then hit Enter, we see the output "O hai, Cansu!"
- What happens when we hit Enter without entering anything at all as a name? We get the output "O hai, (null)!" In our program, we don't actually check that the user provided a valid name as input. This can be a huge security issue and is often how programs and websites are compromised: by not properly handling malformed user input. In general, even if you think that people are truly good at heart,[8] as a computer programmer, you must prepare for the possibility that they are adversaries.
- Question: how is the user's input stored in memory? We won't delve into this too deeply at this juncture in the course, but suffice it to say that a string is stored as a series of characters, each one taking up a single byte of memory. A special character `\0` denotes the end of the string.
- Question: what is the difference between GCC and other compilers? Very little. All of them are responsible for translating source code into binary, although some of them are specific to certain programming languages, processor types, and operating systems. Some of you may have used an *integrated development environment* (IDE) which allows you to compile a program with the click of a button. On a lower level, these IDEs run a compiler similar to GCC.
- Question: how does `make` work? `make` is configured to look for a makefile which specifies the various command-line flags and arguments that we wish to pass to the compiler. We'll begin writing makefiles later in the semester.

[6] We prefer the term naive, thank you.
[7] Cansu is our head TF. Her name is pronounced JAN-soo. Say hi to her. Hi, Cansu!
[8] David's not as nice as he looks, trust me.

- Question: why can we run the `gcc` command without prepending `./`? Because we installed GCC the standard way, it lives in a directory that the computer knows to look in when trying to run any program.
- As a sidenote, take a look at `holloway.c` and see if you can figure out what it does. It was written as part of a obfuscated code contest, which is a bunch of nerds who try to write overly complex, hard-to-read source code.[9]

4.4 A Few Useful Linux Commands

- Keep in mind that all of this will be walked through in the first problem set and there are also resources linked to from the course website, so never fear if you get lost on this part!
- The `ls` command will list out the files in your present working directory. Speaking of which, the command `pwd` tells you what your present working directory is.
- To change directories, type `cd` followed by the name of the directory.

4.5 More on C

4.5.1 `printf`

- As we mentioned earlier, the `printf` function can take many different formatting characters. Just a few of them are:
  - `%c` for `char`
  - `%d` for `int`
  - `%f` for `float`
  - `%lld` for `long long`
  - `%s` for `string`.

  A `float` is a number that can also hold decimal places. A `long long` is a number that can store larger numbers than an `int`.

4.5.2 Escape Sequences

- `\n`, as we've already seen, is the newline character. `\t` denotes a tab character. In general, escape sequences are used to denote characters that might be misinterpreted either by a human or by the compiler. For example, whitespace (e.g. a newline or a tab) blends in with the source code and can easily be represented in shorthand. As well, we need a special character to represent null or nothing. Moreover, to enclose a `string` in C, we make use of double quotes. However, what if we want to include a

[9] SPOILER ALERT: It prints out "hello world!" Also, Snape kills Dumbledore.

literal double quote character within our `string`? We'll need to *escape* it by writing `\"`.

- Another escape sequence worth mentioning is `\r`. This is a so-called carriage return and, in the world of Linux, it denotes bringing the cursor back to the far left. Unfortunately, in the world of Apple, it means something different. In fact, Windows, Linux, and Apple systems all store line endings differently, namely as `\r\n`, `\n`, and `\r`, respectively. At some point in your life, you'll run into a problem that results from this, most often because you opened a text file on a system other than the one that it was saved on.
- Confusingly, the escape character `\` itself needs to be escaped. If you want to print out a backslash character, you must write `\\`.

4.5.3 Mathematical Operators

- Most are self-explanatory, but one you may not have seen before is the modulo operator `%`. This calculates the remainder after division. `11 % 10` gives us `1`, for example. This operator will become extremely useful later in the course.

4.5.4 Types

- C, unlike many other languages (e.g. JavaScript, PHP), has built-in variable types called *primitives*. This is because C requires that you specify the type of a variable when you first declare it so that the operating system knows how much memory to allocate. Here's a short list of primitives in C, many of which you've already seen:
  - `char` (1)
  - `double` (8)
  - `float` (4)
  - `int` (4)
  - `long long` (8)

  In parentheses next to each primitive is the number of bytes of storage allocated for each. As you probably understand intuitively, given that numbers are represented by a finite amount of space, you can't represent every single one of the infinite numbers that exist.
- Another consequence of finite storage space is imprecision. Consider *Office Space* or *Superman 3* where tiny fractions of pennies were stolen and ostensibly not missed.
- A `double` is a `float` that can store larger numbers.

- What's the largest number we can represent with an `int`? Well, there are 32 bits, each of which can hold two possible digits, so the largest number we can represent is $2^{31} - 1$ or approximately 4 billion. If you want to store negative numbers, as well, you must use a `signed int`, which can represent numbers between -2^{16} and 2^{16}.
- For your convenience, we've also created a few new variable types that are available to you when you include the CS50 library:
  - `bool`
  - `string`

  It turns out that "true" and "false" are represented as 1 and 0, respectively. In computer science, one of our goals is to abstract away non-intuitive details like this. So in the CS50 library, we define `true` to be 1 and `false` to be 0.
- Neither is `string` a built-in data type in C. In the CS50 library, we've defined `string` to be a `char *`. We've done this to avoid confusion in these first few weeks because a `char *` is actually what's called a *pointer*. A pointer doesn't store the value of a variable, but rather the memory address of that variable. More on that later.

4.5.5 Operator Precedence

- Recall from grade school that multiplication and division are evaluated before addition and subtraction. This is the idea of *precedence*, which appears in computer science as well. Numerous resources in the recommended reading and online will provide the full order in which C operators are evaluated, so we won't go into more specifics here. Just be aware of it.