

Contents

1	Announcements and Demos (0:00–6:00)	2
2	From Last Time (6:00–14:00)	2
3	From Scratch To C (14:00–24:00)	3
4	Introduction to C (24:00–76:00)	5
4.1	Writing, Compiling, and Executing	5
4.2	Some Jargon	8

1 Announcements and Demos (0:00–6:00)

- 0 new handouts.
- Problem Set 0 is due at noon tomorrow! You need to upload your project to MIT's website and also fill out this [form](#).
- Don't forget about Walkthroughs on Sunday nights at 33 Oxford Street! Sometimes we have trouble getting everyone's ID card activated to get into the building, so if you find yourself locked out, dial 617-BUG-CS50 and hit the secret option 0 to get in touch with one of the TFs.
- Post your questions to [help.cs50.net](#).
- Office Hours will be held tonight. When you arrive, point your web browser to [queue.cs50.net](#), answer a few questions about the reason for your visit, and virtually raise your hand! The CS50 Greeter will then pair you with a TF and your browser will begin to flash when it's your turn to be helped.
- Scribe Notes are here for your viewing pleasure! If you're reading this, then you're already benefiting from my ability to turn David into a pretty PDF document.
- Full transcripts of the lectures are also available alongside the videos. In the interest of saving paper, we don't print many hard copies of the slides and source code used in lecture, but they're available online in advance of class.
- Sectioning will commence this weekend. More on that during Friday's lecture.
- Ever wondered what that big honkin' contraption is in a glass case in the Science Center? It's the [Harvard Mark I](#), a computer used by Harvard starting in 1944. It was capable of doing computations up to 23 significant digits and weighed 10,000 pounds. Multiplication took 6 seconds, division took 15.3 seconds, and a logarithmic or trigonometric function took over 1 minute.

2 From Last Time (6:00–14:00)

- Recall from last time that ASCII is a mapping from binary numbers to alphabetic characters, symbols, punctuation, etc. The letter A, for example, can be expressed in binary as 01000001, or 65 in decimal.
- To hammer this point home, we'll bring eight volunteers on stage, one each to represent the digits of a byte. The rightmost volunteer represents the 1's column and the leftmost volunteer represents the 128's column with the rest filling in for the other columns in between. When a volunteer

raises his hand, it represents a 1 in the corresponding column. After three rounds of this, we can represent the numbers 66, 79, and 87, which spells out the word B-O-W.

- To reiterate our words of wisdom from last week: you're not the only one who feels "less comfortable" with the world of computer science. A few pieces of advice: start your problem sets early and don't be afraid to walk away from them when you get frustrated. It's amazing how many bugs you'll solve just by giving yourself some time and space to breathe.

3 From Scratch To C (14:00–24:00)

- Although it may not seem like it at first, C is actually a relatively simple language. You'll be able to learn it within a few weeks.
- Recall our first C program from last week:

```
#include <stdio.h>

int
main(void)
{
    printf("0 hai, world!\n");
}
```

The blue "say" puzzle piece from Scratch has now become `printf` and the orange "when green flag clicked" puzzle piece has become `main(void)`.

- 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("0 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("0 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 causes “O hai!” to be printed 10 times.

- In C, a loop that increments a variable and announces its value 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";
```

We’ll discuss strings in more depth in the coming weeks, but think of them for now just as words. Here we have an array that stores only one item, the word “Orange.” The first line of code above creates the array and the second line of code above stores the word in the array’s single bucket.

- Know that, by convention, computer scientists start counting from 0. That's why the first and only element in the array above is the 0th element. This is also why last week was Week 0 and you'll be turning in Problem Set 0 tomorrow.

4 Introduction to C (24:00–76:00)

- In our first C program, we wrote a few cryptic words, among them `int main(void)`. We can gloss over this for now, but know that this is telling the compiler that our `main` function (which is comparable to the “when green flag clicked” puzzle piece in Scratch), will return an integer value when it finishes executing. If you've ever seen an error on your home computer that has a number associated with it, chances are that this number is the return value of that program's `main` function in whatever language it was written in. This return value allows a programmer to know *which* failure his program exited with. 0 means everything finished normally. The `void` tells the compiler that our program takes no input. The final cryptic word in this program is `printf`, which is a function that prints words to the screen. You can think of a function as a miniature program that takes input and produces output.

4.1 Writing, Compiling, and Executing

- To write programs in C, we'll need the help of a text editor. For now, we'll just use TextEdit on the Mac, but we could use any number of programs, including Eclipse, Nano, Vim, Emacs, and even Microsoft Word. Once we write our very simple C program,

```
#include <stdio.h>

int
main(void)
{
    printf("0 hai, world!\n");
}
```

we save and exit. What we've written isn't exactly 0's and 1's that the computer can understand, so we need something that will translate the human-readable C language into machine-readable binary. We could use Xcode or Visual Studio or some other compiler for this, but in the interest of standardizing these tools across all the different operating systems used by students, we're instead introducing the CS50 Appliance. As we hinted earlier, this software is a virtual machine that allows you to run an instance of the Linux operating system within whatever operating system your personal computer runs.

- To install the CS50 Appliance, you'll first need to download and install VirtualBox, which allows you to administer multiple virtual operating system instances on your computer. Because computers these days are actually quite overpowered, we can take advantage of their spare resources in order to run virtual machines. When we open VirtualBox and double click CS50 Appliance, a window will open that resembles the desktop of a Mac or a PC, perhaps with different names for menus and icons. Now we can click the CS50 Menu in the bottom left and choose gedit, a text editor very similar to TextEdit on a Mac or Notepad on a PC.
- As before, we'll write the program above, this time saving it on the Desktop as `hello.c`. The `.c` extension is a convention for programs written in C. Within our gedit window, at the bottom, there is a line that begins with `jharvard@appliance (~):` and has a blinking prompt. This is the command line for Linux. The command line allows us to execute programs by typing their names and hitting Enter rather than double clicking them. By default, we've given everyone the username `jharvard`, short for John Harvard.
- At the command line, we'll type the command `ls`, short for list, and see the following output:

```
jharvard@appliance (~): ls
Desktop lectures
jharvard@appliance (~):
```

This tells us that in your home directory, which is comparable to the My Documents folder on Windows or the Documents folder on Mac OS, there are two directories, or folders, named Desktop and lectures. Recall that we saved our program on the desktop, so we need to navigate there in order to compile and run it. To do this, we run this command:

```
jharvard@appliance (~): cd Desktop
jharvard@appliance (~/Desktop): ls
hello.c
jharvard@appliance (~/Desktop):
```

`cd` stands for change directory. Within the parentheses, we are reminded that we are currently in the Desktop, a subdirectory of the home directory which is represented by the `~`.

- Now to compile our program. At the command line, we'll pass our source code file into a compiler named GCC (GNU Compiler Collection) like so:

```
jharvard@appliance (~/Desktop): gcc hello.c
jharvard@appliance (~/Desktop):
```

The fact that nothing happened when we ran this command is actually a good sign: it means there were no errors in compiling the program. Now when we check the contents of our Desktop, we see an extra file has been created:

```
jharvard@appliance (~/Desktop): ls  
a.out hello.c
```

`a.out` is the default name for a program created by GCC. To run it, we'll simply type its name at the command line:

```
jharvard@appliance (~/Desktop): ./a.out  
0 hai, world!  
jharvard@appliance (~/Desktop):
```

Woohoo, our first program ran! Incidentally, the `./` tells Linux to look for the program within our current directory, represented by a dot. If this were a built-in program in Linux, we wouldn't need to tell it where to look.

- `a.out` isn't a very descriptive name for our program, so let's modify our compiler command to name its output more appropriately:

```
jharvard@appliance (~/Desktop): gcc -o hello hello.c
```

The `-o` is a flag or a switch or an option passed to GCC to tell it the name we'd like it to give to its output, in this case `hello`. Now we can execute the same program by running `./hello`.

- For the first of many Linux tricks you'll learn this semester, know that you can press the up and down arrows to scroll through the list of your previously run commands.
- Remembering which options to execute GCC with every time we run it would be a huge pain. Thankfully, there's another program we can use to compile our code: `make`. We run it like so:

```
jharvard@appliance (~/Desktop): make hello  
make: 'hello' is up to date.
```

`make` knows to look for the source code file `hello.c` and turn it into a program named `hello` even without being passed any non-standard options. It also prints us a user-friendly message telling us that `hello` doesn't need to be recompiled because it's already "up to date."

- If we want to recompile `hello` anyway, we need to remove the previous output. We do this with the `rm` command:

```
jharvard@appliance (~/Desktop): rm hello a.out
rm: remove regular file 'hello'? y
rm: remove regular file 'a.out'? y
jharvard@appliance (~/Desktop): ls
hello.c
```

`rm` will prompt us if we really want to delete the files we just specified and when we type `y`, it will actually remove them. Now we have nothing but `hello.c` left in our Desktop directory. When we run `make` now, we get the following:

```
jharvard@appliance (~/Desktop): make hello
gcc -ggdb -std=c99 -Wall -Werror hello.c -lcs50 -lm -o hello
```

As you can see, `make` is actually running `gcc` in the background with a few extra options that we've told it to pass by default. More on these options later. As before, we can run `./hello` to execute our program.

- Question: by convention, directories and executable programs are highlighted in bold by the `ls` command.
- The CS50 Appliance is a full-fledged operating system in and of itself. We can click on the Firefox icon and browse the web within the virtual machine just as we would on any other computer.
- Question: how did `make` know what the name of our source code file was? Because we told it to compile `hello`, it looked for a source code file named `hello.c` by default. In the coming weeks, we'll set up `make` to use aliases so we can compile multiple files when we specify a keyword.
- Question: what would happen if we tried to run `make hello.c`? In short, we'll get an error message saying there's "nothing to be done" for this file.
- Question: do the files we create within the Appliance exist anywhere on our actual operating system? No, not as separate files. They are all housed within a single, large `.vmdk` file. However, throughout the semester, we'll have multiple ways of interacting with this environment aside from this Appliance.
- Know that the Appliance can be run even without an internet connection!

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, which we'll call arguments, and return some output. The first function we wrote was called `main`. In our first program, we passed `'0 hai, world!\n'` as an argument to `printf`. The `\n` is a character which denotes a new line.

- C has a number of *primitive types* built into it. These include `int`, `char`, `float`, and more. `int` is the integer type which stores numbers using 4 bytes or 32 bits of memory. How many numbers can we store using 4 bytes? Each bit has 2 possible values, so in total, we can store 2^{32} , or around 4 billion, numbers. Generally, this means we can store the numbers from negative 2 billion to positive 2 billion. Note that there are serious consequences to this storage space being finite. Once we count up past 2 billion, we're going to run into problems with the `int` type. To store numbers this large, we'll need more bits. This is where the notion of 64-bit integers, namely a `long long`, will come in handy.
- `float` is the floating point type which can store decimals. Here too finite storage space presents a problem. One third represented as a decimal translates into an infinite number of 3's after the decimal point. We obviously can't represent an infinite number of digits with finite storage space, so at some point we're going to have to cut it off. This can lead to serious miscalculations. Think back to the Y2K bug which was caused by using only enough storage space to represent two of the digits in the year rather than the full four.
- 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.

- If we wanted to make our program from earlier slightly more interesting, we could make use of the CS50 Library like so:

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

int
main(void)
{
    printf("Name please: ");
```

```
    string s = GetString();  
    printf("%s\n", s);  
}
```

`GetString` is going to do the work of parsing the user's input and passing it back to us so that we can store it in a variable named `s`. We're then going to pass this variable `s` as a second argument to `printf`, which we'll substitute it in where the `%s` is.

- What's the deal with the `include` lines? These lines tell the compiler to make use of other libraries of code within our program. The *standard library* is `stdio.h`, which allows us to use the `printf` function. Similarly, including the `cs50.h` header file allows us to use `GetString` which is defined in the CS50 Library.
- Now when we compile and run our program, typing "David" when prompted for a name, we get something like the following:

```
jharvard@appliance (~/Desktop): make hello  
gcc -ggdb -std=c99 -Wall -Werror hello.c -lcs50 -lm -o hello  
jharvard@appliance (~/Desktop): ./hello  
Name please: David  
David
```

- Next, let's try prompting the user for a number instead of a name:

```
#include <cs50.h>  
#include <stdio.h>  
  
int  
main(void)  
{  
    printf("Number please: ");  
    float f = GetFloat();  
    printf("%.10f\n", f);  
}
```

Now we're calling the `GetFloat` function and storing its return value in a variable with type `float`. We've also changed the format string that we pass to `printf`. The `%.10f` tells `printf` to substitute in a floating-point value and to print 10 digits after the decimal point. After compiling and running the program, providing the number 0.1, we get the following:

```
jharvard@appliance (~/Desktop): ./hello  
Number please: 0.1  
0.1000000015
```

Weird. We definitely didn't type that. What this demonstrates is the inherent imprecision of floating points. We only have a finite number of bits, so some rounding has to be done in certain cases.

- Don't think that floating point imprecision is a big deal? Check out [this video](#) to be convinced otherwise.