

Section Notes

Week 2

Computer Science 50 — Fall 2008

Written by: Doug Lloyd '09

Week of September 28, 2008

Contents

1	What is computer programming?	2
2	Building Blocks	2
2.1	Declarations	2
2.2	Assignments	2
2.3	Conditional Statements	3
2.4	Boolean expressions	4
2.5	Loops	5
2.5.1	while loops	6
2.5.2	do-while loops	6
2.5.3	for loops	6
2.6	Variables	7
3	Do you “C” what we mean?	8
3.1	main(), Function Types, and Return Values	8
3.2	Header Files	9
3.3	Putting It All Together	9
3.4	Now It’s Your Turn!	10
3.5	Mystery Function	10
4	A Crash Course in Linux	11
4.1	Accessing the Linux Systems	11
4.2	The Linux Shell	12
4.3	SSH	12
4.3.1	A really important note!	12
4.4	The Structure of Linux Commands	13
4.5	Getting Online Help	13
4.6	Linux Files and Directories	13
4.7	Creating, Reading, and Editing Files	14
5	Administrivia	14
5.1	Setup	15
5.2	Assignments	15
5.3	cs50.net	15
5.4	Lectures, Sections, and Code Walkthroughs	16
5.5	Miscellaneous	16

1 What is computer programming?

“Computers are useless. They can only give you answers.” – *Pablo Picasso*

Computer programming is, at its base level, the process of thinking about, writing, testing, debugging, and ultimately executing code on a machine such that a certain task is then automated for you. The world as we know it today wouldn't be possible if it weren't for computer programming. For that reason alone, it makes a lot of sense that you would be interested in taking this class. During the course of the semester you will learn a great many techniques that you can then use to make everyday tasks in your life simpler. Bugged down by factoring those polynomials? Write a program to do it for you! Like being able to instant message your friends but hate the available programs for one reason or another? Write your own application! Just by knowing the building blocks, you can make your computer do some pretty complex stuff! Not to sound cliché, but the possibilities are endless.

In a grand sense, the above quote from Picasso (who, you may recall, gained his fame in quite a different field from computer science) is actually quite accurate. Computers are inherently dumb. They only know how to manipulate things in terms of 1's and 0's. That's it; nothing more complicated than that. But yet at the same time, they are incredibly smart. Given the right inputs and a skilled programmer at the helm, computers can produce in nanoseconds results that it would take humans hours, days, even years to arrive at on their own.

CS50 will teach you how to harness this computational power. But it's up to you to use it effectively. With that in mind, let's begin.

2 Building Blocks

Programs in any language, such as **C**, which we will be using for most of the course, are easily broken down into basic building blocks¹. In the following subsections, we introduce some of the most fundamental, and provide an example of each in both Scratch and C (Don't worry if you don't understand the C code just yet...we'll get there soon!)

2.1 Declarations

A declaration is akin to willing something into existence. Notice that this is not the same as a declaration in English, because stating “I want a new bank account...now!” doesn't create that bank account for you. Though,

```
int my_bank_account;
```

which is a *variable declaration* in C, does create a container called `my_bank_account` for you. Don't worry about that `int` that comes before `my_bank_account` in the variable declaration just yet. We'll talk about that when we get to variable types below.

You've seen variable declarations in Scratch, too. (In fact, we certainly hope you used them in Problem Set 0!) If you click on the “Variables” tab, you get a new menu which gives you the option to “Make a Variable”. Alas, with C you have to make a little more effort to get that variable, but both of these things are equivalent.

2.2 Assignments

Another common type of building block is variable assignment. Let's use `my_bank_account` as an example again. If I said, “I want there to be \$10,000 in my new bank account...now!”, that (sadly) does not put that money in the bank account for you. But in C, something as easy as

¹Unless, perhaps, you're participating in the annual Obfuscated C Code Contest – http://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest

```
my_bank_account = 10000;
```

will put that \$10,000 into your fictional bank account.

In Scratch, variable assignment is very simple as well. You could, from the Variables tab, simply insert the following puzzle piece into your script (we hope that you did something similar!) and it would have the same effect:



It is also possible to combine declaration and assignment for variables in C. That is, the declaration: `int my_bank_account = 10000;` is completely acceptable. It both declares the variable and assigns a value to it. We call this “initialization”.

2.3 Conditional Statements

“If there is something to be changed in this world, then it can only happen through music.”

– *Jimi Hendrix*

A *conditional statement* makes a judgment on information that is given, and then follows a particular course of action as a result of having evaluated that information. There are a few different conditional statements that can be used in C: the `if` statement, the `switch` statement, and the ternary operator (`?:`). For now, we will only touch on the `if` statement since it is familiar to you from Problem Set 0.

In Scratch, you used an “if” puzzle piece similar to this one:



This meant that if the mouse button was depressed, then any of the puzzle pieces that were inside of the arms of the “if”-piece would then be executed. If the mouse button was not depressed, then those inside pieces would not be executed.

Well, in C things are not too much different. We won’t use actual C to demonstrate this example, as it would require us to introduce *Boolean expressions*, which we’ll be doing in just a minute. Instead, we’re going to introduce something called *pseudocode*. Pseudocode is your friend! Especially when you are designing a program from the very beginning and you don’t want to get mired in the details of variable names, complex lines of code, etc...pseudocode is something great to use. In pseudocode, you keep some basic C syntax intact, but you replace a bunch of it with plain English instead. Perhaps it’s better if we just show you an example.

Imagine that we could somehow take the quotation at the beginning of this section and transform it into C pseudocode. Well, it could look something like this:

```
if (there is something to be changed in the world)
{
    it can only happen through music;
}
```

So, if there is something to be changed in the world, then it can only happen through music. But, if there is nothing to be changed in the world...we can’t tell from this snippet of code. All that we know is if (and only if) the condition is true will we execute the code that follows in between the braces.

And what’s up with those braces, anyway? Think about braces in the same way you might think about the arms of the “if” puzzle piece in Scratch. The truth of the condition at the top leads to the execution of everything between the braces. You’ll soon find that you won’t get very far in C without using braces!

The if statement can be extended in two ways. It can be an if-else, or an if-(else if)-else. Examples of both are given below:

EXAMPLE OF IF-ELSE

```
if (a number is less than 5)
{
    do something;
}
else
{
    do something else;
}
```

EXAMPLE OF IF-(ELSE IF)-ELSE

```
if (a number is less than 5)
{
    do the first thing;
}
else if (a number is greater than 10)
{
    do the second thing;
}
else
{
    do the third thing;
}
```

You can see how these extensions of the if statement allow for greater flexibility in the actions we can perform when we judge the truth or falsity of the condition. And how, exactly, do we test the truth or falsity of the condition in C? Now comes the use of the Boolean expressions!

2.4 Boolean expressions

A *Boolean expression* takes one or two arguments (variables, numbers, other...) as inputs and returns either true or false based on what those arguments are and the Boolean expression used to compare them. A couple of these Boolean expressions will probably be familiar to you from basic mathematics, and remain unchanged in C.

(A < B)
(A > B)
(A <= B)
(A >= B)
(A != B)
(A == B)

Pay particular attention to the last of these. Note that to check for equality between two arguments, we use two equals signs (==) and not one (=). Why is this? Recall that when we assign a value to a variable we only use one equals sign. Therefore,

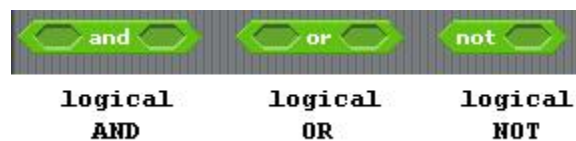
```
my_bank_account = 10000;
```

says that my_bank_account now has \$10,000 in it, while

```
(my_bank_account == 10000)
```

will evaluate to true if my_bank_account has \$10,000 in it, and false if it does not.

In addition to these rather familiar Boolean expressions, you will also be charged with dealing with three more complex Boolean expressions, all of which were available to you in Scratch as well:



And what kinds of things might go in those empty hexagons in Scratch? Well, expressions like `4 < 7` or `mouse down?` or `key F pressed?`. Basically, anything that can be assigned a value true or false. It's also worth noting that, in C, **EVERYTHING** can evaluate to true or false. For example, every integer evaluates to true, except for 0, which evaluates to false. That is, you can say something like:

```
if (4)
{
    my_bank_account = 10000;
}
```

And whenever the computer goes to execute that line of code, your bank account's balance will be set to \$10,000. (It's worth noting that you will never find this particular line of code in any major bank's software.)

Above, we showed the three Boolean expressions in Scratch. How would we represent these in C? If we assume that A and B are expressions that have definite truth value (either true or false), then:

`A && B` `A || B` `!A`

are, respectively, logical AND, logical OR, and logical NOT. Okay, so now that we know what they are, what do they evaluate to?

- Logical AND evaluates to true only in the case where A is true and B is true as well. In all other cases, (`A && B`) is false.
- Logical OR evaluates to true if either A is true or B is true, or both A and B are true. Only when both A is false and B is false does (`A || B`) also evaluate to false.
- Logical NOT takes only one argument and yields the opposite truth value. That is, if A is true, then (`!A`) will be false—and vice versa.

Using these Boolean expressions will be invaluable to you when you need to compare two values and make decisions based on the outcomes of that evaluation!

2.5 Loops

If we needed the computer to count to 10 for us, there are a couple of ways it could be done. We could, for example, say to the computer (in pseudocode):

```
set counter = 0;
counter = counter + 1;
display counter's value;
counter = counter + 1;
display counter's value;
.
.           (7 more times)
.
counter = counter + 1;
display counter's value;
```

But that would be horrific! Imagine having to code that in every time you needed the computer to count to 10? Or 100? Or 1000? And what if you needed the counter to be able to change? You'd need to write a separate program (or introduce a LOT of `if-else` statements) to get the desired results. Well, the designers of C figured you might ask your computer to count to 10 some day. And so they have provided you with three ways of *looping* to make a tedious task significantly simpler. Below, we introduce those three loop styles. We strongly encourage you, based on the templates we provide, to write in pseudocode (or in actual C, if feeling ready) this counting-to-10 example.

As an exercise, we encourage you to look at the Scratch puzzle pieces and try and build the following loops.

2.5.1 while loops

A **while** loop will execute continuously as long as a given condition, provided at the beginning of the loop, is true. In fact, if the condition in the loop is *always* true, then your program might never get out of the loop, and this is what is termed an *infinite loop*. You don't want those in your programs...so if you write something simple and it doesn't seem to be ending...you might have gotten trapped in an infinite loop.

Here is the structure for a **while** loop.

```
while (condition)
{
    stuff;
}
```

It's a good idea, especially if your condition is something like ($x < 10$) to change the value of x somewhere in the body of the loop. Otherwise, you will almost assuredly get stuck in an infinite loop because you will never get out of the loop if x is always less than 10.

2.5.2 do-while loops

A **do-while** loop is quite similar to a **while** loop except for one thing. Notice that it is possible to never enter a **while** loop. Namely, if your **condition** is false, the code inside the loop never gets executed. Now suppose you, for one reason or another, wanted to run through your loop once, and THEN check to see if your condition is true or false. This is what a **do-while** loop permits you to do.

```
do
{
    stuff;
} while (condition);
```

It's also very important to note the semicolon that follows the **do-while** loop. It is part of the syntax for the loop and cannot be left out. This construct will do whatever **stuff** is, and then check to see if **condition** is true. If it is, it will continue to do **stuff** until **condition** is no longer true. If **condition** is false, it simply moves on to the next line of code and will not iterate through the loop one more time.

2.5.3 for loops

The third and final kind of loop you may use is a **for** loop. These loops have significantly different syntax from the previous two but perform the same function. We'll first show you the general form and then explain the different parts:

```
for (initialization ; terminating condition ; modifying statement)
{
    stuff;
}
```

- The *initialization* of the loop usually takes the form of an assignment. For example, assigning the variable i to the value 0. ($i = 0$). We use this initialization along with the other two parts of the **for** loop to control how many times we run through the loop.
- The *terminating condition*, when it evaluates to false, terminates the loop, preventing the code inside the loop from being executed another time. Usually, this is a check to see if some variable that is being modified by the loop, either inside the loop or in the modifying statement, has reached a certain value. (e.g. $i \neq 40$)

- Lastly, the *modifying statement* is executed once the code in the body of the loop has completed running. This usually changes the value that was initialized in the initialization. A common statement to see here is `i++`².

It is also possible to get stuck in an infinite loop when using a `for` loop. Which of these three components, if left blank, would automatically result in an infinite loop?

Now that you know what each of these three pieces means, it's important to note the order in which they are evaluated: First, the loop's *initialization* takes place. Immediately thereafter, the *terminating condition* is checked. If the statement is false, the loop immediately terminates. If the condition is true, then the body of the loop executes. After the body of the loop has executed, the *modifying statement* is run and then the *terminating condition* is checked again. From there, the pattern of *terminating condition–loop body–modifying statement–terminating condition* repeats until the terminating condition is false.

2.6 Variables

Variables are the spice of life in computer science. Without variables, you wouldn't be able to have done most of what we've discussed above—certainly not without a significant amount of hard coding. And, up until now we've assumed that variables are always numbers, specifically *integers*, but in C there are many different forms that a variable can take on. You will likely find, when we do examples, that `int` variables are the most common. When we declare a variable in a declaration, we must give it two things: a *type* and a *name*. Thus, the original declaration:

```
int my_bank_account;
```

Says that you want to create a variable of type `int` (which is an *integer*) that is named `my_bank_account`. Having created this variable, you can now assign any integer value to it that you want.

Integers are not the only type though. For example, it wouldn't be very useful for storing the first letter of a last name, as you may find in software for alphabetically storing student records. In this case, we might want to do:

```
char first_letter;
```

In this case, `first_letter` is a variable of type `char`, a *character* variable, which can be assigned any letter. For example, saying `first_letter = 'a'`; would assign a lowercase `a` to that variable.

The two other major built-in types are `float` and `double` which provide for single- and double-precision floating-point values (which approximate the real numbers). The difference between a single- and double-precision floating-point value lies in how much space the computer's memory uses to hold information on that variable. Obviously, the more space you use, the more information you can store and the more precise your variable's value will be. If you wish to know even more about the difference between single- and double-precision, a quick Google search will probably tell you more than you ever wanted to know.

Also, while `int`, `char`, `float`, and `double` are the four most commonly used variable types, C provides some others as well, and there are also provisions for you to create your own data types...but that's a more advanced topic that we'll get to later in the course.

As a final note, it's worth mentioning that trying to assign a floating-point value to a character variable, for example, could potentially result in undefined behavior. "Undefined behavior" may seem like a vague term, and that's because...well...it is! What you should take it to mean is that it shouldn't be done. That's why data types are there in the first place. Further advanced techniques such as *type casting* can eliminate this problem, but this is also something that will be discussed later on, though we encourage enterprising students to conduct a little bit of research into the concept of type casting to see how it fixes this problem.

²This, you will learn, is shorthand for `i += 1`, which itself is shorthand for `i = i + 1`.

3 Do you “C” what we mean?

After all of this talk about the essential pieces of a C program, wouldn't it be nice to see one put together completely? We'll do that now, looking at one of the simplest programs that you could possibly write in C, “Hello, world!”. “Hello, world!” is a program that simply, upon being executed, prints the words “Hello, world!” to the terminal output and then exits. Following the code, we break it down to clear up any confusion there might be about things you might not have seen before and things we haven't discussed just yet. Here's the code, which we wrote in a file called **hello.c**:

```
#include <stdio.h> // Section 3.2

int main(int argc, char *argv[]) // Section 3.1
{
    printf("Hello, world!\n"); // Section 3.2
    return 0; // Section 3.1
}
```

One quick note before we go any further. Notice those slashes? `//`. Those are *comment delimiters*. Anything that you type after those two slashes, so long as it stays on one line, will be ignored when the code is compiled by the computer into an executable program. For multiple-line comments, use `/*` at the beginning of a comment, and `*/` at the end of a comment. The staff highly encourages you to use comments, both for our sake when you are writing code that gets a little bit complicated, but also for your own sake, in the event that you happen to forget what that particular piece of code does!

3.1 `main()`, Function Types, and Return Values

A *function* is a series of lines of code that may or may not take inputs (*arguments*), performs some sort of manipulation, and outputs a *return value*. This is basically no different from a function that you learned about in high school math. In math, the function:

$$f(x, y) = 4x + 3y$$

when given the arguments $x = 2$ and $y = 3$, manipulates those arguments based on the rules given in the function body, and then outputs the return value to us: 17.

A C function acts in much the same manner. When referencing the names of functions, it is conventional to place a pair of parentheses afterwards. For example, the most important function in any C program is the function `main()`. Why is `main()` so important? Well, when a program is executed, it starts with the first line of code that is inside of the `main()` function. If you wrote a C program that didn't have a function called `main()`, when you tried to compile the code, you would get errors! In general, you can't pass any arguments to `main()`, since that's where the program starts, but `main()` does have two special arguments that can be used, called `argc` and `argv`. But we're getting a little bit ahead of ourselves. Don't worry about these for now, just know that we'll come back to these two special function arguments later in the course.

You'll notice that before `main()`, there is a type, `int`. Just like variables, functions can (and, in fact, must) have types. But what this *function type* indicates refers only to the kind of value that is returned when the function is complete. An `int`-type function will return an integer upon completion. A `char`-type function will return a character upon completion, etc. There is also another type besides those previously discussed called `void`. `void` functions do not return any value at all.

The reason that the last line of this function is

```
return 0;
```

is really out of custom more than anything else. Traditionally, `main()` is either of type `int` or of type `void`, though we recommend that you use `int` for this purpose. And, traditionally, a return value of 0 is used for `main()` if the function did everything that it was supposed to do and exited normally, and a nonzero return value is used for instances where an error occurred and the function had to exit abnormally. Do you have to follow this custom? Well, technically, no. But it is good style to do so, and style is important!

3.2 Header Files

Okay, you ask, what the heck is going on with that `#include` thing at the beginning of this program? Sometimes, for convenience, we want to use functions that other people have already defined for us...but we don't want to have to copy and paste them into our code, because that would make it look clunky. For example, the `printf()` function that is used isn't defined anywhere in our `hello.c` file, so it has to come from somewhere. Well, it just so happens that `printf()` is declared in the *header file* called `stdio.h`. This stands for the "Standard I/O" header file. If you are writing a program that involves taking input from a user at the terminal or printing anything to the terminal window, you will need to `#include <stdio.h>` at the beginning of your `.c` file.

`#include` is what is known as a *preprocessor directive*. The compiler replaces the line `#include <stdio.h>` with the complete contents of the `stdio.h` file. It keeps your code nice and clean and does all of the tedious copy and paste work, so that you don't have to.

Since `stdio.h` contains a description of how `printf()` can be called, we are now free to use `printf()` anywhere we want in this file. This is particularly important for us in this case, since our entire program is dependent on the use of `printf()`. And what does our call to `printf()` do? It takes a string (a series of characters) as its argument and ignores the result.

So, how does the text end up on the screen then? Along with taking arguments and producing return values, functions can also have *side effects*. Printing output to the terminal is a side effect of what `printf()` does. Oh, and that `\n` that follows the text as the argument to `printf()`? That just inserts a *newline*. It's just like hitting the "Enter" key when you're using a word processor. There are many different kinds of side effects, and inevitably we will encounter them as we progress through the course.

Header files come in lots of different flavors. There are some that contain functions to deal with input and output (`stdio.h`), others deal with strings (`string.h`). There is a library that allows you to use the current time in your program, if you need to (`time.h`). We highly recommend the website www.cppreference.com, which has a section dedicated to the C Standard Library header files. Everything you need to know about what functions are contained in which header files should be covered there.

There's a lot more we could say about header files, but we'll wait a few weeks, until you feel more comfortable writing functions of your own.

3.3 Putting It All Together

So now you have this code all written up and ready to go. So, how do you run it to see if it works? We must *compile* our code into an *executable*, and then we can run the program. There is a program that we use, `gcc`, which compiles code into this executable format. Conveniently enough, `gcc` is called a compiler. If we are certain that our code file (`hello.c`) is free of errors and we are ready to take a shot at compiling it, we go to our shell and type in at the command prompt:

```
gcc hello.c
```

If all goes well, and then you `ls` your current directory, you should see a new file called `a.out`. This file is the compiled version of your code! Now comes the ultimate test: Does it work? To test to see if it works, all we need to do is run the file. To do so, simply type at the command prompt:

```
./a.out
```

And if all is well and everything has been done correctly, the words `Hello, world!` should appear and a new command prompt should appear below. How exciting! You've just written and compiled your first successful C program!

Perhaps you don't want the executable file to be called `a.out`. After all, it's not really representative of what the code actually does. Well, `gcc` does provide you with the option to name an executable something different. You just have to use a *flag* during your call to `gcc` to do it. For example, if we wanted to name our new program `hello`, instead, we could do so by typing the following at the command prompt:

```
gcc -o hello hello.c
```

And that would produce a new executable called `hello`, instead of `a.out`.

3.4 Now It's Your Turn!

Take a look at the following function, which prints the alphabet out to the terminal and then exits. Briefly, we'd like to explain that funky argument to `printf()`. Sometimes, when we make calls to `printf()`, we want to print the value of a variable instead of an explicit text string. To do this, however, we need to make a special call. We use the character sequence `%c` to represent a character variable. And then to specify which variable we want, we put it in as an argument to the function. That is to say that:

```
printf("%c", first_letter);
```

will print out whatever the value of `first_letter` is at the time the call is made, and

```
printf("%c%c", first, second);
```

will print out whatever the value of `first` is immediately followed by whatever the value of `second`. Aside from `%c`, there are other symbols that can be used to represent different variable types. For example, `%d` can be used for integers and `%f` can be used for floats or doubles. If you Google `printf()`, you'll quickly find a list of all of the sequences that you could use in `printf()`.

What are the statements in the following function? The conditionals? The Boolean expressions? The loops? Variables? Break down this function into all of the smaller segments and make sure you fully understand how and why this function works. Talk to your TF if you have any questions—they'll be more than happy to clear things up!

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char letter;

    for(letter = 'A'; letter <= 'Z'; letter++)
    {
        printf("%c", letter);
    }

    return 0;
}
```

As is probably apparent, it is legal to cycle through the letters of the alphabet using the `++` operator. If you add 1 to `'A'`, you get `'B'`, etc. (But, be careful! What do you think happens when you go past `'Z'`?)

3.5 Mystery Function

Now we want you to implement a function for us! We've filled in some of the code for you, but there are several numbered blanks. In this function, which is in the file `squares.c`, we want you to make it so that if we compiled the code and then ran the resulting file it would print out the squares of all the numbers from 0 to 10. What would you need to put in the blanks to make this happen? Don't introduce any new variables—just use the ones we have declared at the beginning of `main()`:

```
#include <stdio.h>

int main(int argc, char *argv[])                (1) _____
```

```

{
    int number;                (2) -----
    int number_squared;       (3) -----

    number = __(1)__;         (4) -----

    do
    {
        number_squared = __(2)__;
        printf("%d ", __(3)__); (6) -----
        __(4)__++;
    } while (__(5)__);

    __(6)__;
}

```

4 A Crash Course in Linux

Linux is an operating system that performs many of the same types of functions as other operating systems with which you may be familiar, such as Windows, MacOS, and DOS. An operating system is a complex piece of software whose primary job is to make efficient use of hardware and facilitate communication between applications running on your computer and the computer's hardware. Although you may have never seen Linux before you arrived at Harvard, it is used widely throughout the world. Universities, the Internet, and the computer industry have traditionally shown strong support for Linux.

One reason for Linux's popularity is that it is a good software development environment. Linux has many built-in shortcuts that an experienced user can use to navigate quickly. Since people have been using Linux for a long time, there are many useful programs and utilities that have already been written. However, unlike some other graphical operating systems, the user interface of Linux is often quite terse. There are not a lot of dialogs, icons, and menus to wade through in order to do what you want; many Linux tools have only a command line interface — an interface that you need to type to, instead of navigating with a mouse. While this can be confusing at first, with practice you will soon be able to navigate through the operating system, run programs, and use commands like a pro (we promise!). Linux will not swamp you with confirmation dialogs and warning boxes; instead it only tells you what you need to know in a manner that may seem a little bit cryptic at first, but will soon seem concise and succinct.

Some of the machines in the basement of the Science Center run Linux, which is a member in the UNIX family of operating systems. The distribution of Linux that we use is called Ubuntu, which still uses a command prompt like all Linux systems, but also has a Graphical User Interface (GUI) similar to that of Windows and MacOS. For this reason you may find it to be more user-friendly than other installations of Linux you might eventually encounter.

4.1 Accessing the Linux Systems

The best way to learn the fundamentals of Linux is to try things out. There will be a problem set soon that walks you through various examples, but for now just try to get familiar with the environment.

If you have a personal computer which is connected to the network, you can access the Harvard network in a couple of ways. The first way, with which you may already be familiar, is to use a secure shell, such as SecureCRT, PuTTY, or Terminal. These programs allow you to accomplish tasks on a Linux machine (like writing CS50 programs) while sitting in front of a remote, potentially non-Linux machine, like your PC or Macintosh. As a CS50 student, you will connect to the following computer:

- `nice.fas.harvard.edu` — For course use. NICE stands for New Instructional Computing Environment. (The old system's name was ICE, which just stood for Instructional Computing Environment.) Many tools are available only on NICE, so be sure to use this system for doing course work. If you are doing your CS50 homework, log on to `nice.fas.harvard.edu`.³

There is also `fas.harvard.edu`, but this is for general (non-course) use. Use these machines for checking email or other casual purposes, such as accessing the `ph` database.

One of the first things you may notice is that you won't be able to use your mouse to manipulate the screen!

4.2 The Linux Shell

Above, we mentioned something called a secure shell, and you may be wondering what a shell is. The shell is a program that reads your commands, interprets your commands, and then directs the operating system to execute your commands. You yourself will never need to interface directly with your operating system.

The shell will *prompt* you when it is awaiting a command. The prompt is an indication that the computer is ready for you to enter a command. This is the only visible sign of the shell you are likely to encounter. The shell is, however, one of the most important parts of the system, without which you would be unable to run programs or interact easily with the computer. The shell's prompt is where you will enter your commands to the system, instructing it what commands to execute.

The exact prompt will vary depending on what system you connect to and how your environment is set up. For example, on the Linux Workstations, it will default to `ws01%` or something similar. You will know that you are logged on correctly and have successfully run the setup command described in Problem Set 1 if your prompt looks like:

```
username@nice (~):
```

By convention, prompts often end with a percent sign. Therefore we will typically indicate a prompt by typing a percent sign followed by the relevant command for you to type: `% <command to type>`. Remember that the percent sign is not part of the command or you'll get inexplicable error messages!

4.3 SSH

One of the first things we will ask you to do when you open up your Linux shell is to make sure that you are working on the nice system. If you are working from home, you can log into NICE directly, but from the science center computers, you must SSH onto nice. (SSH just stands for secure shell). To do this, you will type at the prompt:

```
% ssh -Y username@nice.fas.harvard.edu
```

You will then be prompted for your password. You must do this every time you sit down to work on CS 50 problem sets.

4.3.1 A really important note!

If this is the first time you have logged in to your FAS account (or just to make sure it's configured correctly)...type the following in at the command prompt as soon as you've logged in:

```
~cs50/pub/bin/cs50setup
```

This command will configure your account correctly for use with CS50!

³We will actually be changing to a different computing environment soon. It is still useful for you to know this information, as other courses will use NICE. We will provide you with updated information on how to access our new computing environment when it becomes necessary.

4.4 The Structure of Linux Commands

A Linux command is composed of the command name and possibly some *arguments*. An argument is a modifier to the command, which can specify how the command will work or what the “target” of the command is. You can think of commands as verbs, and arguments as objects or adverbs. For example, to send email, you might use the command

```
% pine
```

In this case, the command is `pine` (which is a UNIX-based e-mail program) and there are no arguments. However, if you were in a hurry to send email to your friend, John Harvard, you could type

```
% pine jharvard
```

By specifying `jharvard` as an argument to the command `pine`, you will make `pine` open directly into the email composition screen. You could list multiple addresses one after the other if you desired. Finally, you may simply want to check the messages in your inbox quickly, so you might type

```
% pine -i
```

This will instruct `pine` to open straight into the inbox and list your messages. Note the dash or minus sign (`-`) in front of the `i`. This means that the argument is a special kind of argument commonly called a *switch*, a *flag*, or an *option*. Switch-type arguments are commonly used to switch the command between two modes of operation, and may have arguments of their own. Although the number of options may seem overwhelming, have no fear—Linux was designed by people who didn’t want to look everything up: that’s why there are many online help resources you can use to figure out what arguments a command accepts.

4.5 Getting Online Help

If at anytime you are unsure about the exact capabilities of a command you are using or if you do not know which command carries out certain functionality you can refer to the online manual page. “Man pages,” as they are called, are a great place to begin when you have a question. In fact, if you ask a TF a question that is answered in a man page, do not be surprised if that TF directs you to the man page first.

For example, if you are interested in knowing what the `ls` command does, you can simply type `man ls` at the prompt. In this case, `man` is the command name, and `ls` is the argument telling `man` which manual page to display. This will display the man page for `ls`, which describes how to use the `ls` command and all of the options for `ls`. Man pages contain a lot of information in a small amount of space, so sifting through to the information you need might seem like a daunting task. However, learning to read man pages now will save you time later. Consider what the command `man man` would do...

Once you have opened a man page, you can search the man page by typing `/` followed by a search string. To exit from a man page, type `q`.

If you are feeling adventurous, you may also use the `info` command, which works in a similar fashion to `man`, but may at times give you more detailed information. The choice is yours.

4.6 Linux Files and Directories

Several special names for particular directories provide a convenient shorthand when specifying directory names in Linux:

- `~` (pronounced “twiddle” or “tilde”)

Tilde is the name for your home directory, which serves as a container for your personal files. Within your home directory, you can create subdirectories to organize your information. All files and subdirectories you create on the FAS network will be contained in your home directory or subdirectories of

your home directory. You cannot access the home directories of other users, nor can they access your home directory.

Tilde is useful for giving directions to a file that doesn't depend on what directory you're currently connected to. For example, the command

```
% more ~/.cshrc
```

will display the contents of the file named `.cshrc` in your home directory, no matter what directory you happen to be in at the moment.

In general, a `~` followed by a username represents the home directory of that user. For example, `~cs50` is another name for the home directory of the `cs50` account (a place from where you will be getting a lot of files during the course of the semester).

- `.` (pronounced “dot”)

Dot is the name of whatever directory you happen to be in at any given moment. For example, the command

```
cp ~cs50/pub/distributions/pset6/* .
```

copies all the files in directory `~cs50/pub/ps/src/ps1` into the current directory.

- `..` (pronounced “dot-dot”)

Dot-dot is the name of the parent directory of dot. For example, the command `% cd ..` changes the current directory to be the parent of the current directory.

- `/` (pronounced “slash” or “root”)

Root is the name of the top-level directory. All of the other files or directories in the system are descendants of root. Your home directory (and everyone else's) is a distant descendant of root. You own the files and directories in your directory, but you do not own the parent directory of your home directory or any other of its descendants.

4.7 Creating, Reading, and Editing Files

For CS50 you will need to modify files provided by the course staff as well as create your own files. To do these tasks, you will use a standard Linux text editor, **nano**. Like much of Linux, it is designed by and for programmers and it has many features that will come in handy if you take the time to learn to use them fully at the beginning of the semester.

Starting **nano** on the command line is fairly simple. You can simply type **nano**, followed by the name of the file you wish to modify or create.

```
% nano my_filename.txt
```

We wish to remind you, again, not to dismay if a lot of this technical stuff is confusing to you. There are lots of people here to help and once you get acclimated to the use of Linux, you'll be a pro in no time!

5 Administrivia

Due to the large enrollment of CS50 and the complexity of administering such a course, it is essential that students are familiar with its procedures and policies.

5.1 Setup

- You must have a FAS account. If you do not have an account and do not know how to get one, that should be addressed immediately after section.
- Once you have chosen your password, never share it with anyone else. Do not tell it to anyone, under any circumstances. This is a very, very serious matter. Someone who tells you that they need your password in order to do some administrative or course-related task is lying (or at best, misinformed). Report the person at once to the proper authorities.

5.2 Assignments

- Even if you choose to develop your assignments on your own computer with your own compiler (which is neither recommended nor supported), your assignment must work on the Linux platform, be compiled by the course-specified compiler (`gcc`), and be submitted electronically using the `cs50submit` program.⁴
- Read instructions carefully and do everything the problem sets tell you, regardless of how trivial it may seem. Many students are frustrated when they lose points on quizzes and problem sets for careless omissions.
- A helpful starting point when it comes to starting a design from scratch is just to stop and think for a while. Rushing to a computer terminal to get going isn't necessarily going to translate to actually accomplishing anything. Stop. Think about how you would structure the problem. Break the problem into smaller pieces, get each piece up and running, and then integrate it with the next larger piece. Use pseudocode.
- If you have a question or concern with your performance on an assignment or the manner in which it was graded, please talk to your TF after class or during office hours. Your TF will explain to you why you missed the points or how to correct your mistake. If your work was graded incorrectly, your grade will be adjusted; however, the grading standards are not negotiable. If you are still unhappy with the grading of your assignment after you have spoken to your TF, please see David.
- Lateness policy: Note that the lateness policy for this course is slightly complicated, but reasonable. Please make sure that you understand it, so that you are not surprised later. Each student is given 9 late days. Use them wisely. You could use all of them on one assignment, or you could spread them out and use them on several different assignments. Remember that these days are not divisible, so even an assignment submitted two hours late will require a full late day. Contact the head TF if you have extenuating circumstances concerning late days. Please, as with any aspect of the course, if you have any questions about the exact application of the late day policy ask, do not guess. Late days only apply to coding assignments.

5.3 cs50.net

The course's website, `cs50.net`, always contains the most up-to-date information available on the course. Use it to get information on the problem sets, find extra resources and books recommended by the staff to help you get a jump start. The full schedule of office hours and sections is available. You can also find all of the past lectures in various media formats, in the event that you can't make it to a CS50 lecture. In the near future, you will be able to access the bulletin board, where you should direct general questions on problem sets, as your questions may also benefit other students who simply may not have arrived at the same roadblock. Basically, anything you need, you can find it here.

⁴Again, at least for the time being, until we transition to a new computing framework.

5.4 Lectures, Sections, and Code Walkthroughs

- Attending lectures is essential to keep pace with the course. They will cover most of the material that you will need to complete the assignments.
- Copies of course handouts distributed in lecture are available on the web. If you miss a lecture (or the handouts run out during lecture), you should try to get a copy of the handouts and go over it before the next lecture or section.
- Sections are not intended to be (nor will they be) a regurgitation of lecture. There will be time to clarify difficult concepts, but most weeks they will be extensions of lecture, often covering new topics that won't be discussed by David.
- Section will cover in depth those concepts and topics that you will need to complete the assignment. Section will try to make clear how the topics covered in lecture apply to the current assignment.
- Part of section will tend to focus directly on the assignment. Usually this will include brainstorming approaches to the problem set, giving you hints on subtle aspects of the assignment, and addressing some of your concerns. For that reason, it is in your interest to come to section having read (if not started on) the current problem set. However, the primary goal of section is to provide you with new tips, tricks, and insights, not to walk you through the problem set. That's what the code walkthroughs are for!
- There will be weekly code walkthroughs on Mondays from 6:00-7:30pm at 52 Oxford Street. If you are stuck on where to start, this is the place to go. Line-by-line, we will go through the problem set. It is **STRONGLY** encouraged that you attend these weekly code walkthroughs if you feel you may have difficulty with this problem set. These are also available online! If you come to a TF the night before the problem set is due and ask where to start, don't be surprised if they direct you to watch the code walkthrough in full.

5.5 Miscellaneous

- Check the CS 50 Bulletin Board found on the course website. Sign up for an account and post questions anonymously. TF's will be monitoring this board and will answer posted questions. In addition, you may look at questions posed by your fellow classmates. This is generally a great resource for both getting started or getting past your bug.
- Textbooks are available at the Coop, for those that feel that they would like additional reading.
- CS 50 web site: <http://cs50.net>
- Don't cheat. There is a fine line between collaboration and plagiarism. The course has developed mechanisms for detecting inappropriate collaboration and will be using them. The consequences are severe.
- Getting help in this course is extremely easy. Here are several of your best resources:
 1. Check and/or post to the CS50 Bulletin Board!
 2. Go to TF office hours in the Science Center terminal room.
 3. Use virtual office hours when they are announced.
 4. Call, email, or make an appointment with your TF.
 5. Go to David's office hours.
 6. Attend the weekly code walkthroughs.
 7. Go to another TF's section to supplement your own.

- 8. Form study groups (but be mindful of the course collaboration policy).
- 9. Get a tutor from the Bureau of Study Counsel.
- There are many TF's on staff, each of whom holds at least two hours of office hours in the terminal room in the Science Center basement each week. One advantage of office hours is that the TFs are familiar with the common pitfalls of the current assignment. You will also get to meet many of your classmates and all of the teaching fellows. The TF's are there to help you, so do not hesitate to utilize them as a resource. In addition to office hours in the terminal room, you are encouraged to make appointments with your TF should you have general concerns about the course or your performance. Office hours are designed to help you tackle problem sets and your final project, but your TF will be happy to meet with you to discuss any other questions or concerns that might fall outside of a specific assignment.

Photo # NH 96566-KN First Computer "Bug", 1945

9/2

9/9

0800 Machine started

1000 stopped - machine ✓

1300 (032) MP-MC $\left\{ \begin{array}{l} 1.2700 \quad 9.032 \ 847 \ 025 \\ 9.037 \ 846 \ 995 \ \text{correct} \end{array} \right.$

(033) PRO 2 $\left\{ \begin{array}{l} 2.130476415 \\ 2.130676415 \end{array} \right.$


correct

Relays 6-2 in 033 failed special speed test in relay. 11.00 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multy Adder Test.

1545  Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1630 Machine started.

1700 closed down.

Relay 2145
Relay 3371

Grace Murray Hopper's first "computer bug", 1945