Section Notes 8

CS50

1 TCP/IP

TCP/IP stands for **T**ransmission **C**ontrol **P**rotocol / **I**nternet **P**rotocol. In computer networking, a *protocol* is a set of rules governing the way computers talk to one another to make connections and to send and receive data. The Internet protocol (IP) defines how smaller networks are woven to together to create larger ones, including "the" Internet. TCP is a widely used higher-level protocol that allows applications to communicate reliably and efficiently. TCP/IP is the combination of the two protocols: TCP running on top of IP.

1.1 IP

IP, or **I**nternet **P**rotocol, is a protocol that defines how communication should be handled from computer to computer. Each computer is given a unique address, called an IP address, which identifies that computer on the network. When one computer wishes to send information to another computer, it encapsulates that information within one or more *IP packets*, which are then sent on the Internet. Information from the sender is typically split into smaller chunks in order to reduce the penalty for losing a packet on the Internet. However, since each IP packet incurs some memory overhead, there is a trade-off between packet size and data reliability.

So how does the packet reach the desired computer? Ideally, the destination computer would be physically connected to the sender. However, this is usually not the case. Instead, relay points *route* the IP packet to the destination computer by relaying the messages between several intermediate locations.

IP relay points are also known as *IP routers* or gateways. When an IP packet is sent from a computer, it arrives at an IP router. The router is responsible for sending the packet to its destination, either directly (if it is connected to the destination computer) or indirectly (by sending it to the next router in the path).

It is important to note that IP is a *connection-less* communication protocol. For almost every transmission, the exact routing path is unknown to both the sender and receiver. IP routers are only responsible for the next 'hop' in a message's path, resulting in two important benefits:

1. Each link in the network can be used to handle packets from many different sender-receiver pairs simultaneously.

2. Since the route each packet takes doesn't matter (as long as they eventually reach their destination), IP packets with the same source and destination can follow different routes through the network. Routes may change in response to varying network parameters, such as traffic volume or router/connection errors.

1.2 TCP

TCP, or **T**ransmission **C**ontrol **P**rotocol, is a protocol that defines how communication should be handled between two applications on two different computers. TCP works on top of the IP layer, meaning that it doesn't care how the other computer is reached, just how to reach a specific application on that machine.

In order to identify a specific program on a machine, the program is assigned a *port number*, which is simply an integer. This port number, paired with the machine's IP address, can be used to uniquely identify that program on the Internet. Thus, no two programs on the same machine may have the same port

number. As with IP packets, using TCP incurs some additional overhead per data chunk, since the sending and receiving programs' port numbers also need to be recorded. Some ports are also standardized across machines. Web servers normally communicate the Hypertext Transfer Protocol (HTTP) using port 80; and typically a machine's FTP service will run through port 21.

1.3 TCP and IP working together

Together, TCP/IP handle the various steps involved in transmitting data from a program on one computer to a program on another computer across the network (or Internet):

1. When a program wishes to send data, TCP is responsible for breaking the data into smaller chunks (packets), and communicating the packets to the machine's network software. It also adds a TCP header (port numbers and other TCP-specific information) on top of each packet.

2. IP is responsible for routing the individual packets from the source machine to the destination machine, using their respective IP addresses. This information is attached to each packet (as an IP header) before they are sent out.

3. IP routers on the network use the information in the IP header to determine where to relay each individual packet that they receive. Packets are routed this way until they reach their final destination.

4. When a packet is received on the destination computer, TCP looks at the TCP header on the packet to determine which program it belongs to. Since routes for individual packets may differ (even though they originated at the same sender), packets might arrive out of order. TCP is also responsible for reordering the packets when they arrive, before handing them off to the appropriate destination program.

2 HTML and XHTML

HTML is well-known for being the language foundational to the development of the World Wide Web. Its name stands for *HyperText Markup Language*. "HyperText" just means text that contains links to other text, i.e., other pages, or other sections of the same page. A "markup language" is simply a language that displays textual information while allowing us to package a little more information about the text. We do this by introducing "markups", which are also known colloquially as "tags". We'll come back to this idea momentarily, after giving a brief example of some XHTML syntax. For the purposes of this class we will stress the use of XHTML. More on the differences between HTML and XHTML below.

2.1 XHTML Syntax

Just like when introducing a new programming language, when introducing any new markup language, one of the first things we learn is how to write a "Hello, World!" document description. The following XHTML syntax, produces a web page consisting solely of that text:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
     <head>
          <title>hello, world</title>
     </head>
     <body>
          hello, world
     </body>
</html>
```

Notice how the markup language is allowing us to convey extra information about the text using the markup tags. The outermost `<html>` and `</html>` indicate that everything in between them is well formatted XHTML. Everything between the `<title>` and `</title>` tags is interpreted to be the title of the page, and will display as the title of the window. The text in between the `<body>` and `</body>` tags (which we omitted in the first example) is then recognized as being the text that goes in the display window, the actual content of the web page.

One of the best ways to learn XHTML is to read the "source" of a page. By right-clicking on a web page and selecting "View Page Source" you can see all of the code that forms the backbone of the page. As an example, check out the source of Doug Lloyd's web site (`http://www.people.fas.harvard.edu/~lloyd`), which is written entirely in HTML (not XHTML, more on this below). The syntax on this page is far from complicated.[1] Looking at what the actual page looks like, and then reading the code behind it should give you some idea of how, for example, one writes tables, includes images, writes hyperlinks, and centers text, among other things, in HTML. The list of tags in HTML is quite numerous. If you want to practice writing your own personal web site in HTML to host on the Harvard servers, you certainly can.[2] There are numerous tutorials available on the Internet that go into great detail about all of the tags that HTML allows in its current version.[3]

2.2 XHTML vs. HTML

As mentioned above, we will be using XHTML in CS50. But if you've glanced at the "Hello, World" example above (XHTML) and Doug's web site (HTML), you might not have noticed the difference. XHTML, in fact, is quite similar to HTML. It stands for *eXtensible HyperText Markup Language*, and what that basically means is that it is HTML of a specific dialect that conforms to XML standards. XML is another example of a markup language, and is unique in that it allows users of the language to define their own tags similar to the ones used in the HTML code examples above. An XML document is considered "well-formed" if every "open" tag has a corresponding "close" tag.

Similarly, an XHTML document is considered "well-formed" according to the same rules. There are a few minor stylistic differences, also. A well-formed XHTML document typically begins with a signifier known as the "document type declaration" that comes at the very beginning of the document:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

What this signifier does is indicate that this document conforms to the XHTML 1.1 standard. There also exist the XHTML 1.0 standards, and XHTML 2.0 is currently in development.

Both of the code examples given for HTML would be considered valid under the syntax rules specified for XHTML (after adding the DTD). However, Doug's page would not; let's show you an example of a place in that code (which we can look at by checking out the page source) that does not conform to the HTML standard. In the line that produces the picture of Eliot House, the code is as follows:

```
<img src="http://www.fas.harvard.edu/~liberm/Images/
 Eliot%20House/Eliot%20house-Night400x300.jpg" alt="Eliot House"><br>
```

In this line, there are two violations of XHTML syntax that aren't a problem for regular HTML. Neither the `<img ...>` nor the `<br>` tag has a corresponding closing tag. In HTML we refer to tags such as the image and linebreak tag as "empty", which means that they are self-contained and don't require a closing tag. Not so in XHTML however; remember that each open tag requires a close tag. There is a way to "short-circuit" this for the linebreak tag: we can re-write it instead as `<br />` or `<br/>`, which serves as both an opening and a closing tag. In general, this looks a lot better than writing `<br></br>` when all you want is a linebreak!

[1] If you want a more complicated example, check out the source behind Wikipedia's own article on HTML (which is actually in XHTML, but as we will soon see, the two are quite similar), `www.en.wikipedia.org/wiki/HTML`

[2] Check out `http://www.fas-it.fas.harvard.edu/node/186` for instructions.

[3] A particular favorite can be found here: `http://www.w3schools.com/html/`

One other notable example of non-conforming XHTML syntax in Doug's web page is one of the first lines of text in the page:

```
<p><b><i>Hello...and welcome to my website.</b></i></p>
```

Here, the paragraph tag (another example of an empty tag in HTML) is properly closed. In fact, all three of the tags are properly closed. But the problem with this line that makes it not conform is that the bold and italic tags are closed in the incorrect order. The most recent tag to be introduced in an XHTML document must be the first one to be closed. The line would be properly formatted, then, if it instead looked like:

```
<p><b><i>Hello...and welcome to my website.</i></b></p>
```

A short list of common errors can be found at Wikipedia's article on XHTML.[4] Seeing these errors and the proper formatting should make it clear what exactly the mistakes are and how they are (usually) very easily corrected.

One final thought: It has taken you many weeks to become comfortable with writing code in C. Don't get frustrated if trying to get your head around learning these new languages becomes challenging at times. Just be aware of the resources available to you, and never hesitate to ask for help. Learning a new language is never an easy thing.

3 Cascading Style Sheets

When the Internet began to be widely used in the mid-1990s, HTML was the only standard for designing the look and feel of web pages. And it simply did not offer enough control for its users. Thus, shortly after the World Wide Web Consortium was founded in 1994, Cascading Style Sheets (CSS) burst onto the Internet scene with the help of their creators Hakon Lie and Bert Bos. Style sheets had been part of the HTML plan all along, but without the standardization of the World Wide Web Consortium, style sheet interpretation was browser specific.[5]

3.1 Basic Syntax

CSS provides us with the power to override or add to the meaning of HTML and XHTML tags. To start off with a quick example, if we wanted the background of our web page to appear blue, we can simply create the following style sheet:

```
body
{
     background: blue;
}
```

Here we have `body`, called a *selector*, that corresponds to our `<body>` HTML tag. This means that the following styles will apply to all the HTML inside the `<body></body>` tags. After our *selector* is a left curly-brace followed by a sequence of semi-colon separated *declarations* followed by a right curly-brace. This is the basic syntax for CSS. Now, the background of our web site in between our `<body></body>` tags will appear blue.

There are many *declarations* that can help you add effects to your web pages. The following is simply a short list of basic commands:

`border: style color width`

— This provides a border around the HTML that appears within the tag corresponding to the CSS selector. Note that the three arguments can be in any order. Examples:

[4] http://www.en.wikipedia.org/wiki/XHTML#Common_errors

[5] Still, however, you might notice that your style sheets will act differently in Mozilla Firefox than they do in Microsoft Internet Explorer. This is why many web developers will preview their sites in a multitude of browsers before publication.

```
border:  3px solid black;
border:  solid thin blue;
```

`color: [keyword | #6-digit hex | #3-digit hex]`

— This will change the color of the text. Examples:

```
color:  red;
color:  #0000ff;
```

`font-family: [family-name | generic-family]`

— This will change the font style of the text. Example:

```
font-family: Arial;
```

`font-size: [absolute-size | relative-size | length]`

— This will change the size of the text. Valid values for absolute-size are: `xx-small | x-small | small | medium | large | xx-large`. Valid values for relative-size are: `larger | smaller`. And length is just an integer value. Examples:

```
font-size:  x-small;
font-size:  3pt;
```

From the following, we could give our HTML body a red background, blue text, a white 10-pixel thick dotted border, with Times New-Roman font of size 24:

```
body
{
     background: red;
     color: blue;
     border: white 10px dotted;
     font-family: "Times";
     font-size: 24pt;
}
```

3.2 Embedding CSS in XHTML

Before we dive into more complex features of CSS, let us go over how to include style sheets in our HTML. You can do this in one of two ways: plant the style sheet directly into your `.html` file, or create a separate `.css` file. In CS50, we prefer the latter.

To illustrate the former, we can recreate our "Hello, World" page by placing the `body` selector's properties within a `<style>` tag in the HTML code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
     <head>
          <title>hello, world</title>
          <style type="text/css" media="all">
               body
               {
                    background: red;
                    color: blue;
```

```
                    border: white 10px dotted;
                    font-family: "Times";
                    font-size: 24pt;
               }
          </style>
     </head>
     <body>
          hello, world
     </body>
</html>
```

Now we will get a hideous looking web-page that says "hello, world". Note the `type` and `media` attributes; these are the standard settings, but if you were to replace `all` with `print`, it would only apply this new style to our web page when somebody tried to print it. This feature is often used when a site has a lot of images and a lot of text, but might like to allow its users to print the text without printing the images. (There are visibility tricks you can play with CSS as well to hide the contents of certain HTML tags from appearing on your site.)

The latter, and preferred method is to store the style sheet in a separate file, say `styles.css`, and include it within the `<head>` tags as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
     <head>
          <link href="styles.css" rel="stylesheet" type="text/css" media="all" />
          <title>hello, world</title>
     </head>
     <body>
          hello, world
     </body>
</html>
```

3.3 CSS Classes

So far in this brief introduction to CSS you've seen the basic *selector*, which is just an HTML tag. Another useful kind of selector is a *class selector*. Here's the definition of a class called `field`:

```
.field
{
     padding: 5px;
     text-align: right;
}
```

The `.` is new syntax here, introducing the class name `field`. We cannot use `field` like a standard HTML tag, as in `<field>Hello</field>` to get a padded area of our web page that has text aligned to the right. Instead we will write:

```
<body class="field">
    Hello
</body>
```

So now the body of our web page will be padded by 5px and have all its text aligned to the right. How can this help us? Well, if we want the same style to be used for multiple HTML tags, but not all of the HTML tags in our document then we can simply just re-apply the

`class="field"`

attribute rather than have a CSS selector for each HTML tag we want to affect.

To review, in order to make a *class selector*, we simply create a style sheet declaration whose name is `.name`, where the `.` specifies that we have created a class and *name* can be any alphabetic character string.

4 PHP

4.1 Introduction to PHP

As we've mentioned above, HTML/XHTML are markup languages. A static XHTML web page is only as powerful as a Microsoft Word document. With XHTML, you can have fancy text in different fonts, colors, and formatting. At the end of the day, though, you still only have a static web site that can only be updated if it is changed manually. You cannot use any kind of logic (i.e., `for` loops, `if` statements, etc.), and your source will always stay the same unless you go in and change it yourself.

Imagine you want a web site that can update itself without the webmaster coming along and manually changing the XHTML code. Think of the Office Hours page on the CS50 web site, which rotates the schedule so that the soonest office hours are always at the top of the web page. It would be ridiculous to have someone sit around and manually change the XHTML on this web page every hour! However, if we only had static XHTML, that's what we'd be forced to do to accomplish this behavior. Therefore, we need a dynamic aspect to our web pages, which is where PHP comes in handy.

When we talk about writing PHP in the context of web development, we almost always are talking about a script that will run and will eventually generate XHTML code. That is to say, the only purpose in the PHP script's life is to print out XHTML code, which gets embedded in the proper place on a web page. We can look at a slightly more dynamic and interesting page than the "hello, world" page from before:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Cool Dice Page!</title>
    </head>
    <body>
        You just rolled two dice.  <br /><br />
        Your first roll was: <? echo rand() % 6 + 1 ?>. <br />
        Your second roll was: <? echo rand() % 6 + 1 ?>. <br />
    </body>
</html>
```

Notice that the page looks very similar to a static XHTML page. However, there are a few key differences. One difference is that we'd name the page with the extension `.php` rather than `.html` to signify that this page runs PHP scripts. The other important difference is that we can embed dynamic code in the middle of our standard XHTML code. We've done that where we wrote:

```
<? echo rand() % 6 + 1 ?>
```

This runs a very simple program that generates a number between 1 and 6 to simulate our roll of a die (and then prints that result out). Note that every time you refresh the page, it will re-run the scripts, and you'll get different dice rolls. `<?` demarcates the start of a PHP script and `?>` demarcates the end of one. You are welcome to have several scripts in the same page—just insert them wherever you need them!

There's a lot more to PHP, and we'd encourage you to use `http://www.php.net` as a reference. You should also look at the sample code we give you in the problem set as a starting point. If you have access

to the PHP code of other existing web sites, that is an excellent way to learn more about PHP.[6] If you get stuck, by all means shoot an email to your TF (or, better yet, post to the CS50 bulletin board!), and we'll be more than happy to help you.

4.2 $_GET, $_POST, and $_SESSION

There are several ways of getting information into a web page. One of the most common and easy to understand is the GET method. Quite simply, when using the HTTP GET method, variables are passed in via the site's URL. For instance, if we wanted to pass in two variables, foo and bar, to our example PHP web site, we could do the following:

```
http://www.example.com/index.php?foo=test&bar=test2
```

Then, in our index.php script, we could access these variables by doing the following:

```
$foo = $_GET["foo"]; // This would have value "test".
$bar = $_GET["bar"]; // This would have value "test2".
```

Passing arguments in the URL is not always a good idea. If we wanted to submit our user name and password to a web site, for instance, it would not be a good idea for the user name and password to both be passed in plain text in the URL:

```
http::://www.example.com/authenticate.php?username=cs50&password=ilovecs50
```

The HTTP POST method is a slightly more secure way of passing this information around. The place that the HTTP POST method most commonly comes up is through using XHTML forms.

Consider, for instance, the following form:

```
<form action="print.php" method="post">
    name: <input type="text" name="name" />
    age: <input type="text" name="age" />
    <input type="submit" />
</form>
```

When the submit button is hit on this form, the variables "name" and "age" would be passed via the HTTP POST method. Then, in our other script, "print.php", we could catch those variables using the following method:

```
$name = $_POST["name"]; // Would contain the string that was typed into the name field
$age = $_POST["age"];   // Would contain the string that was typed into the age field
```

One upside of the POST method is that we do not see all of the variables in plaintext in the URL. One downside of the POST method is that a person cannot bookmark a certain query—they have to fill out the form all over again if they want to see the target page again.

Finally, $_SESSION is useful for storing "global" variables that you want to persist as long as a certain visitor is at the site. For instance, if you want a site that users can log in to, you'll probably want to remember who they are logged in as for their whole visit. This can be done by storing something in the $_SESSION variable:

```
$_SESSION["uid"] = 123;
```

Then, in another PHP script, you can check who they are logged in as:

```
$uid = $_SESSION["uid"]; // this will hold 123
```

This method works as long as the user has cookies enabled. The variables themselves are not stored in your browser cookies, but a cookie is used to keep track of who is who. (If there are simultaneous users, you need a way to keep track of which user you are servicing requests for!)

[6] Note that you cannot see the PHP code of any web page by viewing page source from a browser—you'll only see the XHTML code that is generated after the PHP script runs. You'll need to find a friend (or a TF) who wrote a web site and is willing to show you their code.

5 SQL

5.1 Using SQL

SQL is a database system, which sits "underneath" your web site and holds the data that your web site uses. A variety of things can be stored in these databases. One common thing to store is a list of user names and their corresponding passwords, so that your web site can validate whether or not a certain username/password should be allowed to log into the site.[7]

The largest unit of storage in SQL is a database. Generally, for each different project you are working on, you will only have one database. When you connect to the correct SQL server, you will also need to connect to a certain database.

Inside that database, you may have several tables. Tables can be thought of a 2-dimensional spreadsheet with rows and columns. The columns are decided beforehand (i.e., if you have a "users" table, you might decide that the things you want to store are UID, username, password). When you are designing the table, you must decide how many columns there will be. Then, after you've created the table with a certain number of columns, you can add rows into that table. You can add rows dynamically; so, for instance, you might continuously add rows over the course of the web site's life as people register with your web site.

There are several important things that you need to be able to do with tables, which are briefly introduced below. This is by no means an exhaustive list, but certainly a good place to start. If you want to do something slightly different from anything described here, you probably can. Best to google it first, and then ask a TF if you can't figure it out after a google search.

5.2 Creating a Table

This can be done with the CREATE TABLE command while running the command line version of MySQL. However, remembering all of the syntax for CREATE TABLE can be cumbersome and frustrating. Therefore, we encourage you to use another program called phpMyAdmin to create your tables. Once in phpMyAdmin, you will be asked how many columns you want to create, which should be an easy question to answer if you've thought about the design of your table. The next thing you'll need to decide is the type that will go into each column. You'll see a barrage of options, which we encourage you to read about. We'll highlight some of the most important types to understand here:

`VARCHAR` — This is a variable-length string that must have a maximum number of characters specified. With a variable-length string, SQL won't waste memory. I.e., if your max string length is 255, but the string you put into the table is only 10 characters long, SQL will use 10 bytes of memory, not 255.

`CHAR` — This is a fixed-length string that must have a fixed number of characters specified. With the CHAR type, SQL will waste memory. I.e., if your fixed string length is 255, but the string you put into the table is only 10 characters long, SQL will use the full 255 bytes of memory! This can be good, though, for performance reasons (i.e., time performance) if you're storing something like FAS user names, where you know they cannot be longer than 8 characters, so you can set that max string length relatively low.

`BOOL` — This is a boolean. It can hold true or false.

`ENUM` — This is your own type. You need to specify what values the enum can be, and then you can only put those types into this column.

`TEXT` — This is designed for a much larger chunk of text. For instance, if you want to store someone's comment in your table (and comments could get very long!), you'd probably want to make the column for the comment itself a TEXT comment.

[7] A secure site wouldn't store the actual passwords, but would rather store the hash values of the passwords. Those of you who did the hacker edition of the crypto assignment may appreciate why this is a more secure design decision.

`INT` — Just as in C, we have an integer that stores values in the range from -2^{31} to $(2^{31}-1)$. INT's are often used to store the "primary key" (see below) of a table.

Other important considerations: you'll want to select a "primary key" for your table. This is usually a good idea for somewhat complex reasons that are beyond the scope of the course (if you're interested, take CS165, which studies SQL in much more depth!). Your primary key column should be a column, or a combination of columns,that has unique values. So, for instance, in our "users" table, we'd probably make the primary key the UID, because it is reasonable to require that each username be unique from the others. You can use the primary key to unambiguously refer to a single row, since it's unique for each row. This comes in extremely handy.

One last thing you'll need to decide is whether or not each column is allowed to be NULL. For our users table as we've discussed so far, it's probably reasonable to force each of those columns to be non null. It's especially important to force the primary key column to be non null (think about why)! However, if we added a column like "userprofile", it might be reasonable to leave that NULL until the user fills out his/her profile.

5.3 Table Operations

5.3.1 Getting Information from a Table

One of the most common things you'll need to do with a table in your database is get information out of them. After all, if you couldn't do that, what would be the point of having the tables in the first place? The keyword we use on the command line to do this is SELECT. However, the SELECT syntax is pretty complicated, so pay close attention! (What is explained here is a small subset of what you can do with SELECT—there are many fancier things to be done, but you probably won't need them right away. If you do need them, check out google or ask a TF!) At any rate, here's the basic skeleton for SELECT:

```
SELECT <column, or columns separated by commas, or * for all columns>
FROM <table, or tables separated by commas>
WHERE <condition>
ORDER BY <column name>
```

Let's look at an example. Say we had the following two tables:

```
users
+---------------+------------+-------------+
| username      | password   | fullname    |
+---------------+------------+-------------+
| cs50stud      | h4ck3r     | CS50 Stud   |
| malan         | djmftl!    | David Malan |
+---------------+------------+-------------+

comments
+-----------------+-----------------------------+
| poster_username |  comment                    |
+-----------------+-----------------------------+
| cs50stud        | Gee, I'm a pretty big stud! |
| malan           | I'm not a stud :(.          |
| cs50stud        | Sucks to be you!            |
+-----------------+-----------------------------+
```

Say we wanted to collect all comments made by cs50stud. Here's the command we might use to do that:

```
SELECT comment FROM comments WHERE poster_username = 'cs50stud'
```

This would output the following table:

```
+-----------------------------+
| comment                     |
+-----------------------------+
| Gee, I'm a pretty big stud! |
| Sucks to be you!            |
+-----------------------------+
```

We start with all rows in the comments table, but then the WHERE condition helps whittle down the rows to the ones we actually want (i.e., the ones made by cs50stud). Notice that we are only pulling out the comments themselves here. If we wanted all columns, we'd do the following:

```
SELECT * FROM comments WHERE poster_username = 'cs50stud'
```

```
+-----------------+-----------------------------+
| poster_username | comment                     |
+-----------------+-----------------------------+
| cs50stud        | Gee, I'm a pretty big stud! |
| cs50stud        | Sucks to be you!            |
+-----------------+-----------------------------+
```

You'll probably find that most of the time you just want to do a SELECT * query, because most of the columns will be interesting to you and worth keeping around. But of course it depends on what you're trying to do!

Let's do something slightly fancier with SELECT. Let's say we wanted to be able to post the user's full name alongside their comment. This sounds harder! Well, we can select information from two tables, by performing what's called a cross product (known as a `JOIN` in SQL). This first query will perform the simplest cross product between the two tables:

```
SELECT * FROM users, comments
```

```
+----------+----------+-------------+-----------------+-----------------------------+
| username | password | fullname    | poster_username | comment                     |
+----------+----------+-------------+-----------------+-----------------------------+
| cs50stud | h4ck3r   | CS50 Stud   | cs50stud        | Gee, I'm a pretty big stud! |
| cs50stud | h4ck3r   | CS50 Stud   | malan           | I'm not a stud :(.          |
| cs50stud | h4ck3r   | CS50 Stud   | cs50stud        | Sucks to be you!            |
| malan    | djmftl!  | David Malan | cs50stud        | Gee, I'm a pretty big stud! |
| malan    | djmftl!  | David Malan | malan           | I'm not a stud :(.          |
| malan    | djmftl!  | David Malan | cs50stud        | Sucks to be you!            |
+----------+----------+-------------+-----------------+-----------------------------+
```

The cross product has combined all possible rows in the first table with all possible rows in the second table. If the first table has n rows and the second table has m rows, then the cross product will make a table with n*m total rows. More generally, if there are three tables you are crossing, with n,m, and p rows, the cross product will make a table with n*m*p rows. Clearly, not all of these rows are even "valid" in a sense. In the current application, it seems that rows like:

```
+----------+--------+-----------+-------+--------------------+
| cs50stud | h4ck3r | CS50 Stud | malan | I'm not a stud :(. |
+----------+--------+-----------+-------+--------------------+
```

are somehow "invalid" because they combine a user on the left who did not even post the comment in question. In order to deal with this problem, we'll need to be careful with our WHERE clause:

```
SELECT * FROM users, comments WHERE username = poster_username

+----------+----------+-------------+-----------------+------------------------------+
| username | password | fullname    | poster_username | comment                      |
+----------+----------+-------------+-----------------+------------------------------+
| cs50stud | h4ck3r   | CS50 Stud   | cs50stud        | Gee, I'm a pretty big stud! |
| cs50stud | h4ck3r   | CS50 Stud   | cs50stud        | Sucks to be you!             |
| malan    | djmftl!  | David Malan | malan           | I'm not a stud :(.           |
+----------+----------+-------------+-----------------+------------------------------+
```

Sure enough, this filters out the comments we're interested in. Now, to put the finishing touch on, we only wanted the full name next to the comment:

```
SELECT fullname, comment FROM users, comments WHERE username = poster_username

+-------------+------------------------------+
| fullname    | comment                      |
+-------------+------------------------------+
| CS50 Stud   | Gee, I'm a pretty big stud! |
| CS50 Stud   | Sucks to be you!             |
| David Malan | I'm not a stud :(.           |
+-------------+------------------------------+
```

Another important thing to realize is that you may have ambiguities with the column names. For instance, if instead of "poster_username", we had just named that column "username" (which would have been perfectly reasonable, since there is no other "username" column in comments), we would have gotten into trouble. The way we would have solved that is as follows:

```
SELECT fullname, comment FROM users, comments WHERE comments.username = users.username
```

Notice we've included the table names to help disambiguate "username"!

One last thing that did not come up in these examples, but is certainly important, is that the WHERE clause can have several different conditions. This looks like

```
SELECT fullname, comment FROM users, comments
WHERE username = poster_username AND username = 'cs50stud'
```

Can you figure out what this query would produce?

5.3.2 Inserting a Row into a Table

You can use phpMyAdmin to insert into a table, but it is important that you know how to do it at the command line, so you can write code to do so. Here's the basic syntax:

```
INSERT INTO <tablename> VALUES (colname = colvalue, ...)
```

Here's a concrete example:

```
INSERT INTO users VALUES (username = 'cs50tester',
                          fullname = 'CS50 Tester', password = 'test')
```

This should be pretty straightforward. Make sure that you specify values for things that are not allowed to be NULL.

5.3.3 Removing Row(s) from a Table

If you wanted to delete the user 'cs50stud' from the users table, you could do the following:

```
DELETE FROM users WHERE username = 'cs50stud'
```

This will delete all rows from the users table that match the condition. In this case, it would just delete the one row, but you need to be careful because it will delete as many rows as match the WHERE clause! (Once you delete them, they will be gone! No recycle bin here!)

5.3.4 Modifying Row(s) in a Table

If you already have a row in your table, but you want to change a value, you will need one last piece of syntax to modify the row. Perhaps this is best shown by example. Let's say you had the users table mentioned above, and David Malan has realized that he made a typo when typing his first password, and he wants his new password to be "djmftw!". The way to do this update is the following:

```
UPDATE users SET password = 'djmftw!' WHERE username = 'malan'
```

If there were more than one row in users where the username was 'malan', it would update ALL of the rows, not just one! However, this is not a concern for this table. You can use a more complicated WHERE statement to make sure you're only modifying the rows you mean to be modifying!