Table of Contents

[Introduction (If nothing else, read this) 3](#_Toc429068515)

[Things You’ll Need 4](#_Toc429068516)

[What this system will and will not do 5](#_Toc429068517)

[How it works, in a nutshell 5](#_Toc429068518)

[Limitations 6](#_Toc429068519)

[The basics of managing points 7](#_Toc429068520)

[The Responsibilities of the VP: “What you have to do” 7](#_Toc429068521)

[phpMyAdmin 7](#_Toc429068522)

[Modifying the points system 9](#_Toc429068523)

[The basics 9](#_Toc429068524)

[Changing Point Values or Restricted Status 10](#_Toc429068525)

[Adding a new event 11](#_Toc429068526)

[Getting Advanced: Adding variable points 12](#_Toc429068527)

[Final Exam: Adding an entire points category 13](#_Toc429068528)

[Changing the Approval Page Behavior 14](#_Toc429068529)

[But What about …? 14](#_Toc429068530)

[Full Documentation 16](#_Toc429068531)

[AdminApproval.php 16](#_Toc429068532)

[AdminApproval2.php 16](#_Toc429068533)

[AdminPage1.php 17](#_Toc429068534)

[AdminSetDates.php 17](#_Toc429068535)

[AdminSetDates2.php 17](#_Toc429068536)

[AdminTotals.php 17](#_Toc429068537)

[AdminTotals2.php 18](#_Toc429068538)

[CheckPoints.php 18](#_Toc429068539)

[CheckPoints2.php 18](#_Toc429068540)

[default.php 19](#_Toc429068541)

[footer.php 19](#_Toc429068542)

[FormP1.php 20](#_Toc429068543)

[FormP2.php 20](#_Toc429068544)

[FormP3.php 20](#_Toc429068545)

[GetPoints.php 21](#_Toc429068546)

[Header.php 22](#_Toc429068547)

[Points system manual.docx 23](#_Toc429068548)

[Styles.css 23](#_Toc429068549)

[Appendix A: 10 minute Crash-Courses in HTML/Javascript/CSS/PHP/SQL 24](#_Toc429068550)

[HTML 24](#_Toc429068551)

[CSS 26](#_Toc429068552)

[JavaScript 26](#_Toc429068553)

[PHP 27](#_Toc429068554)

[SQL 30](#_Toc429068555)

# Introduction (If nothing else, read this)

Welcome, and congratulations on your election to the vice presidency of Chapin. Alternatively, welcome to the only engineer/computer science major in Chapin this year, who has been tasked by the VP to dissect and master the points system so as to impose the VP’s new points-based edicts. Either way, welcome!

The job of vice president has traditionally been [aside from the spring quarter rush of housing and the traditional responsibility for all organized Chapin “vice” ;) ] a fairly easy position; I have therefore gone the extra mile to make sure your job remains as easy as possible.

This guide **is not** intended to be read all at once. It’s not even really intended to ever be read entirely by one person. It’s organized so that you can most easily find what you need, and skip over the documentation you don’t.

If you have no plans to change the points system at all, this section along with a quick scan of “What this system will and will not do” and “The Basics of managing points” should be enough to get by for your term. If you or the exec board wants to change the points system by adding new events or changing point values or restricted status, you’ll have to dig a little deeper – read up on “Things you’ll need” and find the relevant sections for what you’re trying to do under “modifying the points system.” Finally, if you’re comfortable with your coding skills, and want to make major changes to how the site functions or handles data, you’ll want to cross-reference with the “full documentation” for the details as to how the site is organized and how it functions.

As I’ve said, I’ve tried to make this guide as easy-to-read as possible (with the exception of my occasional dry wit, which is at best an acquired taste); however, since this will (hopefully) be around years after I graduate, slowly changing over time, I can’t guarantee that everything will be exactly as described. If you plan on making changes, do be kind to future VP’s and document your changes here or within the code comments.

So I wish you good luck for another good year, and may the points be ever in your favor.

Best,

Jonathan Sammon  
Chapin’s “Resident Engineer” 2013-2015  
Northwestern Class of 2017  
8/19/15

# Things You’ll Need

I don’t need any weird engineering software…. Do I?

You should have been given access to the Chapin points system Github, which contains all the files used in the webform. As of the writing of this guide, this should include:

* 7 PHP files beginning with “Admin,” which control administrative functions of the site
* 9 PHP files which control basic, public functionality of the site, including a header and footer
* 1 Template PHP file
* 1 CSS style sheet
* 1 image (the Chapin crest)
* This guide

Github is an extremely common program/website that manages code projects like this one; If you’re going to be making any changes to the code, I **strongly** suggest you download the Github desktop client and learn how to use it; in particular, it offers you the ability to easily collaborate on code projects, and gives you the ability to “branch” code so as to leave a functioning copy of everything while making and testing new changes. The website can be found [here](https://github.com/).

Besides the Github, you should have also separately received a document with all of the login info (I don’t recommend putting that on Github, where it’s accessible online. Security is a good thing). You’ll need this to approve and view points, as well as to manage the site.

You won’t need any compilers or any other software to run the code itself. All of the code is [an unholy mix of] HTML/PHP/SQL/Javascript (in rough order of significance to this project), which are all web-based languages; when you upload them to the site, the webhost and/or the user’s browser will do all the interpreting. I do however strongly recommend downloading Notepad++ [here](https://notepad-plus-plus.org/); editing code through this will highlight syntax and automatically apply formatting to code to make it **much** easier to read compared to a basic text editor like Microsoft’s notepad.

Both Github and Notepad++ are free programs and are familiar to most programmers.

# What this system will and will not do

This website won’t become self-aware, will it?

### How it works, in a nutshell

The first few paragraphs here give a brief understanding of how the site works under the hood. This is useful information, but if you’re impatient, you can skip down to the last few paragraphs, where I have bolded the critical information.

For the points submitter, the process is very simple. The first page of the main form just requests the name, NetID, and points category. The next page is generated based on the category selected, and requests the specific event (options are different based on the category selected), the date, additional comments, and, depending on the category, may have a number field for points or hours, if relevant. The final page confirms the data has been submitted.

All other sections of the code work based on that simple list of Name/ NetID/ Category/ Event/ Date/ Info/ Points list, which also tracks a few other things, like approval status and restricted status. Whenever the form is submitted, this raw submission is recorded in a database table. When a user checks their points, it searches the raw submissions list for approved records in that time period, and sums the result over the three quarters of the current school year.

On the Admin end, you can approve records (which directly edits that same list to mark records as “approved” or “rejected”), and you can generate a list of all names/NetIDs/points totals and subtotals (which is basically just doing the same a checking a single record, but for all of them)

**Here’s the important thing:** The master list makes heavy use of the restricted status, the approval status, and the date of the event (not to be confused with the timestamp, which is just for additional info). **Dates are important!** Records will not show up at all if you’re viewing the wrong date range, or if the points were submitted for dates in the summer. The school year being viewed will always be displayed when viewing the master list or individual records.

Restricted points are automatically capped at 20 per person per quarter. In general, any points above the 20 point cap, as well as any points that were rejected or have not yet been approved will not show up in most reports. You will see warnings on the single NetID lookup page, but *no warnings are built into the final points list generator*. So be sure to finish approving points before creating an official list.

Also note that **all records are tracked by NetID**. This means that if someone’s NetID is typo’ed or otherwise wrong, there will be a separate record created. However, all NetIDs are forced lowercase, so capitalization will not cause problems. All NetIDs are also truncated to 6 characters. Names are not tracked *per se*: The most common name for a given NetID is what shows up for “name.” Therefore inconsistent spelling or capitalization of names will not be a problem, as long as the 6 character NetID is consistent.

### Limitations

**Security:** This system was built on the good-faith assumption that abuse, manipulation, and direct attacks of the system would be rare to nonexistent. Therefore, the focus was on simplicity and ease-of-use than on privacy and security. The most important thing is that there is **no verification** of NetIDs: Anyone can submit a points record for anyone, if they know their NetID. In the same manner, any person’s points totals can be viewed, if you know a person’s NetID. (The master list of all points totals, however, is not publically available). In the future, it may be beneficial to consult ASG (which used to provide student group webhosting and hosted the previous points system) or NUIT about including a Northwestern login functionality to the site for more conventional security.

Note that the admin tools for the points system are in a password-protected directory, which is created through the webhost

**Flexibility:** My goal again was simplicity; there are no full-function tools for editing or viewing specific submissions within the website. The phpMyAdmin tool, however, is available for viewing the raw database, and allows the use of SQL to perform more advanced operations and sortings. This tool is discussed later in the next section.

**Dates:** I was forced to strike a balance between manual controls and automated controls when sorting records by date; there are a few peculiarities of the system to be aware of. When viewing individual totals, the date is chosen automatically by the current calendar date; a general user can only see their points record for the current school year. The master list, however (accessible only by the admin) can be created for different years, if need be.

The quarter that points fall within is determined solely by the 4 dates supplied to define a year (fall start, fall end, winter end, spring end). So a point record for “winter” means it’s dated sometime after fall ended but before winter ended, etc. A point record that falls after spring but before fall will generally be ignored.

**Fractional Points:** Don’t do it. The database expects integer point values. If fractional point values are requested, they will be rounded before being sent on. Significant changes would be required to add this.

# The basics of managing points

How can I do as little work as possible?

### The Responsibilities of the VP: “What you have to do”

At the bare minimum, these jobs must be done by the VP:

1. **Approve Points:** Do this semi-regularly, and encourage Chapinos to submit their points regularly. Don’t wait until the end of the quarter. Note that you can mass-approve records (“Approve” is automatically selected for each record, no matter how many there are), but this is strongly recommended against. (If you ever want to back out of approving records, you can safely hit your browser’s “back” button without submitting anything. However, your approval progress will be lost.)

Note that only points submissions that have neither an approved or rejected status will show up in the points approval page. To change a previously submitted record, you will need to use phpMyAdmin, explained below

1. **Generate a master list:** You’ll need this for housing selection, and possibly for determining voting rights. The list will give you the quarter breakdown of points for everyone who has submitted in a given year, ordered by total points. *Be sure to approve/reject all pending points first!*
2. **Set the quarter dates:** This is important! If you don’t set the quarter start/end dates, the system will not be able to properly generate records. This only needs to be done once a year, and may have already been done by the VP before you. Do it again anyways – It’s important.
3. **Field questions/complaints:** When viewing an individual record, (x points rejected) will be visible under the record if any have been (there’s a similar notification for pending points). You should know how to pull up raw records in phpMyAdmin if a complaint ever arises, since the submitter can’t see what specific record was rejected. Use of this tool is covered below.

The first three tasks can all be completed through links on the points Admin homepage. For anything more complicated, you’ll have to learn a little bit about the tools available to you. In this section, I’ll cover only phpMyAdmin, which is likely to be most useful for resolving minor issues.

### phpMyAdmin

First, some terminology. This tool is poorly named, because it doesn’t deal much with PHP. phpMyAdmin is a database viewer, and gives you basic access to the mySQL database in which the points data is stored. SQL (not to be confused with mySQL [*sorry, I didn’t name these things]*) is a standard language for interacting with any database, like MySQL (other databases include Oracle and Microsoft Access, which you may or may not be familiar with).

This tool is very straightforward to use. Log in to the site management page (<http://members.000webhost.com/login.php>). Click “Go to C-panel,” and you should see a list of all the tools available for managing the website. Scroll down to “Software/Services” and select “phpMyAdmin.” Then click “Enter phpMyAdmin.”

On your left you should see the name of the database (a7686212\_Points), with the three tables it contains (Points\_Summary, Quarter\_Dates, and Raw\_Submissions). Along the top, you should see tools like “Structure,” “SQL,” “Search,” etc.

The only table you’ll likely ever have to edit is Raw\_Submissions (Points\_Summary is always emptied and regenerated when you create a new master list through the Chapin website, and records in Quarter\_Dates can be directly overwritten by using the Chapin website, so they both should never need manual editing.) So with that in mind, click on “Raw\_Submissions,” then click “Browse” along the top. You should see a list of every points submission since this site began; to make life easier, you may want to click on “timestamp” or “date” to sort by those categories. When you find a record you want to edit, you can use to green pencil icon to do so; this will let you change any field of the entry; use this to correct typos, change points amounts, change approval status, or whatever.

If you’re looking for a very specific or hard to find record, the search tool will help you narrow the pool of records. Note that the search tool is basically just generating SQL queries; you can also create your own queries manually if you know some SQL.

I strongly recommend against emptying the table or even deleting individual records, even older points records. The database should be able to handle an unlimited number of entries, and it’s always better to have a record of submissions than to make a habit of deleting things. At the very least, point records should be retained for around 2 years.

Please do not drop tables. (“dropping” a table means deleting it, completely). This WILL break the system. Dropping individual records is ok, but recommended against, as explained above. Emptying a table won’t necessarily break the system (unless you empty the dates table. That will break things until new entries are submitted), but you’ll lose all data, so just don’t do it. Thanks.

phpMyAdmin also allows you to export to a variety of formats including Excel; this may be helpful in some cases.

# Modifying the points system

…I’m going to have to learn some code, aren’t I?

In past sections, I’ve talked generally about what this site can do and how to use its basic functionality. Here, I will be giving more specific instructions for how to modify the actual PHP/HTML code that creates the site itself. Since I will be referring to specific file names, comments, and code sections, I cannot guarantee this has remained or will remain consistent over time; I once again encourage future VPs to update these sections to remain accurate after any changes are made to the code.

The goal in this section is to help you identify the specific parts of the code required to change a certain functionality. I will be talking only minimally about the overall structure and functionality of the code. Hopefully, this will make the code easy to modify **with little coding experience**.

The following sections are semi-cumulative, and progress from simple tasks to more complex changes. If a later section seems to be missing information, you might benefit from reading the earlier tasks and see if that gives some insight. I’ve done my best to point out where sections my build upon earlier ones.

With that in mind, let’s begin.

### The basics

To start, make sure you’ve read over the section “Things you’ll need.” You’ll definitely want to download Notepad++ to make writing code much easier (make sure the language is set to PHP – the program should automatically apply different colors to different parts of the code for ease of reading). Once you’re finished editing the code, you should sync your changes to Github (I strongly encourage branching the code or otherwise making a separate copy of everything so that you can always revert all changes and return to the old, functional code if something goes wrong).

To upload new code to the site, login to the site management page as in the previous section. This time, look for the file managers (there are two). Both should give you the ability to upload new files to the site, view the files there, etc. Don’t worry about deleting old files – uploading a file of the same name will automatically overwrite the existing copy. [NOTE: I’ve experienced errors with the first file manager available; it seems to error out once whenever I login. Closing it out and re-entering the file manager seems to always resolve the issue]. Also make sure you’re uploading to the right folder: Public code goes in the Public HTML folder, administrative code goes in the Admin folder within the Public HTML folder (anything in the Admin folder will be password-protected). Do not upload to the base folder, which helpfully has a file named “DO\_NOT\_UPLOAD\_HERE” to remind you. **Do not** delete any file you don’t understand.

The following sections will require some basic code experience. I’ve tried to simplify everything as much as possible, but you should still understand the basics of what is going on. Appendix A gives a brief description of syntax and function to help, but you should also feel confident googling for specific constructions or doing a few basic code tutorials. If nothing else, learn to recognize comments:

In HTML, a comment will appear as

<!—Comment -->

In PHP (Anything between <?php and ?>), a comment will appear as

// comment

By default, Notepad++ will highlight those in Green.

In this guide, I will write specific code snippets in Monospace Font (traditionally used and easier to read for code) to set it off from the rest of the guide. It will usually be also set off as a new paragraph.

In general, when I write a value in “quotes” or ‘quotes’, the quotes should not also be entered or changed from how they are in the code already. Single quotes and double quotes are slightly different in PHP; sometimes they are interchangeable and sometimes they aren’t, even though I use them mostly interchangeable in the text of this guide itself. So in general, don’t change the quotes just because of how I wrote them here.

If you ever want to understand the purpose of a given file, a brief comment at the top should explain. The Full Documentation section of this guide will provide (slightly) more detailed descriptions of what the code does as well as how it does it.

### Changing Point Values or Restricted Status

Here, I start with the simplest modification: Changing the number of points an event is worth. My system for assigning points isn’t entirely intuitive, but it should be easy to maintain with a little guidance.

The hypothetical example here will be to make “Attending a Fireside” worth 4 points, not three, but make the points restricted.

Open up “FormP3.php”.

Most of FormP3.php is taken up by a very long switch statement that mirrors one on the previous page. It should start with this:

switch($event) {

and has many statements that look like this:

case 'ac1':

$points = 3;

$event = 'Attending a fireside';

break;

Here, you’re lucky – “Attending a fireside” happens to be the first entry, so you don’t have to search. If you’re struggling to find an entry, CTRL-F is always useful. Now, you would just change $points = 3; to instead assign 4 points to the event. To make it restricted, you would add the statement $restricted=TRUE; (There’s no need to have any statement concerning $restricted if the points are unrestricted; points are assumed unrestricted unless assigned otherwise)

So now this block of code looks like this:

case 'ac1':

$points = 4;

$event = 'Attending a fireside';

$restricted = TRUE;

break;

Be sure that you don’t accidently erase the break; as this would cause it to fall through to the next case after executing the assignments, changing all the values to those of the next event and generally causing a bunch of errors that are really hard to debug.

### Adding a new event

Say that the exec board is instituting a policy of awarding points for “Offering sage advice to a freshman.” We’ll say that that falls under philanthropy, and is worth 1 restricted point. And while I also encourage you to point out to your exec board that this is a strange thing to award points for, I will also show you how to add this in.

Open up “FormP2.php”. Most of the file is take up by a large switch statement that begins with

switch($category) {

This part of the code generates the different options for each points category, so when a user selects “Academic,” they only see the Academic events listed as possible to select. Most of the code here is pure HTML, with a little bit of PHP controlling the actual functional switch.

For each category, you should see one or several <option> tags. Let’s take a look at a few:

case 'Philanthropy':

?>

Event: <select name="event">

<option value="ph1">Chapin Philanthropy</option>

<option value="ph2">Non-Chapin philanthropy</option>

<option value="ph3">DM participation or fundraising</option>

<option value="ph4">Dancing on Chapin's DM team</option>

<option value="ph5">DM event (trivia, etc)</option>

</select>

(Note: The ?> is very important; it controls what code is php and what code is html. Do not change this.)

Simple enough. Each event has a few options; each option is assigned a unique alphanumeric code which is eventually passed to the next page. In the case of philanthropy, there are 5 events; to add another, we just need to add another <option> right after the last

<option value="ph6">Offering Sage Advice to a Freshman</option>

Note how we added a unique code (ph6) for this option, following the same format (first two letters of the category, plus a number for its position of the list of options). If you don’t create a code, you’ll have an error where both this option and the option for which the code is the same will both create the same record. Note also that this should be put directly after the last option on the list, before the </select> tag.

Now that that’s added, the form will properly display the option. However, the rest of the code doesn’t know yet how to interpret or assign points to this new event. So we’ll have to edit one other file, “FormP3.php,” which in coding parlance is the “Form Handler” script for Page 2. As seen in the above section on changing point values, this is where the point value and type is set.

So open up “FormP3.php.” You’ll once again see a large switch statement, which for the most part mirrors the one on page 2. Each case in the switch will look like this: (This should look familiar if you read through the previous section)

case 'ac1':

$points = 3;

$event = 'Attending a fireside';

break;

It should hopefully be pretty obvious that you’ll have to add a new case statement to account for the new option. So scroll down to the philanthropy events (this isn’t strictly necessary, you can put the new case anywhere in the list, but we’ll try to keep it organized), and after the case statement for ph5, add your own for the new code:

case 'ph6':

$points = 1;

$event = 'Advice for a Freshman';

$restricted = TRUE;

break;

This should of course come after the break; statement for the case before, and before the case statement for the event after it; if this isn’t observed, you’ll see errors.

Notice also that the $event text here doesn’t have to be exactly the same as the <option> text on the page before. This is the benefit of the strange code system on which the switch is organized.

From here, everything should be handled properly by the rest of the code.

### Getting Advanced: Adding variable points

A few times, the point value to be awarded isn’t fixed. This happens with student projects, points petitions, and hours-based philanthropy, currently. In case you need to add a similar event, let’s take a look at a few of the differences.

In “FormP2.php”, we return to our familiar switch statement full of options. Let’s look at philanthropy: below all the <option> statements you should be familiar with by this points, you’ll see a few other form elements. There’s a description field, which has slightly different text per category. Below that, most cases break and move onto the next, but philanthropy, along with “other,” has an additional field.

Hours of Philanthropy <i>(if applicable)</i>

<input type="number" name="points" size="2"> <br /> <br />

Pretty simple; an input which accepts a number, which is stored under the name “points”. Why is hours of philanthropy stored as “points”? It was just easier to have one flexible number field which alternatively means hours when for philanthropy, but points when for other. *That* weird naming convention *is* my fault. Sorry.

Note that not all philanthropy events use this field. If a submitter enters points but selects an event which has a set point value, this will be ignored.

On the other side (ie, on “FormP3.php”), let’s look at how hours of philanthropy are handled:

case 'ph1':

$points = round(2\*$points);

$event = 'Chapin philanthropy';

break;

In my code, $points actually holds the value from this flexible number field. Usually this is undefined and roundly ignored (it is usually overwritten in the case statement anyways), but here, $points does have an initial value – it’s the number of hours of service. Since points are awarded at 2 per hour, we multiply $points by two, round it (it shouldn’t be possible to enter fractional hours anyways, but nothing is foolproof), and then store the result as the new value of $points (*side note: If you aren’t familiar with code, it may seem odd that points set equal to an expression involving itself. In code, statements like “$i=$i+1” are in fact perfectly valid. In that case, the value of $i is incremented 1 and saved; the simple statement $i+1 has a value, but next time you call $i, its value will not have changed)*.

In the case of points petitions and student projects, the statement omits the 2\*, since the points are 1:1 with what is entered in the field.

### Final Exam: Adding an entire points category

So say your exec board, after instituting points for sage advice and now drunk with power, decides to make some major changes. Tribute given to the president will now earn you 2 unrestricted points, and tribute to a lessor exec member will earn you 1. Since at this point they would struggle to call this “philanthropy,” we’ll make it a new category: “Tribute.”

To start, we’ll have to take a look for the first time at page 1 (“FormP1.php”), since this is where points categories are selected. Towards the bottom you’ll find once again a <select> element populated by a bunch of <option>’s. All you need to do is add another option, just like before:

<option value="Tribute">Tribute</option>

Note that there’s no weird code this time; the value is “Tribute.” This value is case-sensitive and you will need it for the next page, so make sure you keep it simple. Note that it does not have to match the text between the <option> and </option> tags exactly, as long as the value text is the same on this page as on the next.

Now move on to “FormP2.php,” which should feel like home at this point. Let’s add a new case:

case 'Tribute':

?>

Event: <select name="event">

<option value="tr1">Major Tribute</option>

<option value="tr2">Minor Tribute</option>

</select> <span class="error">\*</span> <br /> <br />

Description:<br />

<input type="text" name="info" size="40"> <br /> <br />

<?php

break;

This will of course come after the break;statement for case ‘other’, but before the final case statement, which is default:.

So what’s going on here? First off, the case statement reads the value of the option selected. Since we gave our new option a value of “Tribute,” that’s the case value given. Again, this needs to exactly match the value given to the option on Page 1. Otherwise they won’t match, and this code won’t run. (if nothing matches, the default: code will be run. So you’ll probably figure it out soon). Then we have our <select> element for the event, which contains two <option>s (Major Tribute and Minor Tribute). Each of these options is given a unique ID code, “tr1” and “tr2”, just as before. We also have a description field (you could change the Description: to read whatever you would like. Similarly, it might be appropriate to change Event: to perhaps Variety of Tribute: or something similar; this text is just the text displayed for these form elements.)

Finally, the observant reader will point out that I haven’t explained the ?> … <?php. Because I’m trying to keep this guide as simple and user-friendly as possible, I’ll just say that the position and inclusion of that is very important, but for technical reasons that aren’t important here (it has to do with marking what code is PHP and what code is HTML, which gets complicated quickly). Read up on Appendix A or the Full Documentation for this file if you’re really curious.

### Changing the Approval Page Behavior

Most of the admin code doesn’t need to be updated to handle minor changes to the point system. However, the behavior of Admin approval page can be tweaked for preference, if it becomes an issue.

When designing the Admin Approval page, I deliberately made the choice to automatically select “approved” for every record, assuming that only minimal oversight is needed for approving points, and that the vast majority of records would be approved.

However, this leads to two downsides: First, it is therefore very easy to accidently mass-approve a large number of records. Secondly, there is no way to approve only some points, and to save the rest for later; no matter how many records there are, you must do them at once.

The easy way to fix this is to start off with neither radio button selected. The handling script on the next page will therefore ignore any record not explicitly checked off, solving the problem.

To do this, find this comment:

// If you don't like the possibility of accidentally approving a ton of records,

// you should remove [checked=\"checked\"] from the next line

And do what it tells you to do.

### But what about …?

If you’re read over this entire section, and are still trying to figure out how to replicate or make minor modifications to an already existing element in the form, you should from here be able to work out roughly how everything fits together. Find elements that are similar, copy, paste, and edit them to give you the options that you need, and experiment. I believe in you.

If you want to make major changes to how things work, like adding some sort of “semi-restricted” points classification or automatically rejecting any points submitted on Tuesdays, you’re going to have to read up on PHP, MySQL, and HTML, and make your own changes. The documentation I provide in the final section should help a lot with understanding how the code is structured, how and why it works, but you will have to determine the best way to edit it to your needs.

# Full Documentation

Wait… Why did you just call the die() function? Is that bad?

This section gives the full documentation on how each file works alone and with other files. Most of this will be fairly dry and technical, and all of it requires a decent knowledge of code to make use of, but the information can be useful information if you’re doing large-scale changes to the site. Please keep this updated accordingly. This is organized by file, so if you only need information on one file, this is the place to go. Information on the methodology of each file can be seen as a more-or-less “step-by-step” walkthrough of what the code is doing (It’s basically pseudocode). To use this documentation, I recommend having both the code and this document open at the same time; you should be able to see the correspondence between the intent described here and the physical code used there.

Also note that if you’re looking to write an entirely new file, “template.php” give some of the code common to each page. You may also want to see some of the notes on “AdminPage1.php” if you’re writing a file in a subdirectory (like the admin folder), as some of the include code needs to change accordingly.

Files are organized alphabetically. Generally files that go together are named the same, with subsequent files being appended with “2,” “3,” etc. A few pages such as “default” or “AdminPage1” act as hub-pages, with links to several other action pages.

Let’s begin:

### AdminApproval.php

**Purpose:** This page generates a list of unapproved points records, along with a form with a simple “reject”/”accept” radio button for each record. “AdminApproval2.php is the handling script for the form; that page is responsible for actually updating the records.

**Methodology:**

* Connect to mySQL, and select all records that have not yet been approved
* If any record is found, do the following:
  + Create a table with headings for most of the information associated with a points record, plus a column for a form element
  + Create a form, which overlaps the table
  + Loop over each record. Echo back most of the information retrieved into the table. Also echo two radio buttons for each record with a name the same as the index for the given record, and automatically select “approved”. Also save the highest and lowest index overall.
  + Echo an end to the table and a submit button
  + Save the max and min index in $\_SESSION, so it can be accessed on the next page.
* Otherwise, display a generic error
* Close out the connection

### AdminApproval2.php

**Purpose:** Handle the form from the main admin approval page. Edit the records in MySQL. Link back to the Admin homepage.

**Methodology:**

* Retrieve the indices saved in $\_SESSION.
* Connect to MySQL
* Loop over the index range retrieved above. For each index number, check if there’s a record in $\_POST for the given index. If so, create and run an SQL query to set the status of the record of that index number to the status recorded in $\_POST under that index.
* Also in the loop, keep track of how many records are being processed. Outside and after the loop, echo this back for confirmation that the proper number of records have been updated.
* Link back to the Admin Homepage.

### AdminPage1.php

**Purpose:** Offer a convenient page with links to all admin functions, as well as additional useful internal and external links. Contains no PHP of substance – mostly HTML.

### AdminSetDates.php

**Purpose:** Provides a form for the points admin to set quarter start and end dates for a given year. This form only needs to be submitted at most once a year.

**Methodology:** This is just a very simple form. Each date entry field has a unique ID which is used to run some JavaScript to provide a “datepicker” tool.

### AdminSetDates2.php

**Purpose:** This is handling code for the form on the previous page. Here the dates are entered into the database table.

**Methodology:**

* Retrieve all our data from $\_POST
* Connect to MySQL
* Check to see if there’s an existing record for the year submitted. If so, mercilessly delete it. We don’t want duplicate records for a year.
* Insert the collected dates into the table
* Echo confirmation and offer a link back to the Admin Homepage.

### AdminTotals.php

**Purpose:** This form leads into the master points list. On this page, an admin can enter a date to check past or current totals sheets.

**Methodology:**

* Get the current month and year. If it’s July or later, we assume it’s fall quarter, and keep the year. If it’s earlier than July, we assume it’s winter or spring, and thus the “year” is one less than the current. This keeps the convention of a school year’s records being stored by the year of fall quarter.
* Create a form with two elements: a “year” textbox and a submit button. Give the textbox the value of the current year – this makes it the default value, but this can be changed by the admin when visiting this page.

### AdminTotals2.php

**Purpose:** This file is responsible for sifting through the points database and determining a list of valid NetIDs, and corresponding point totals, as well as subtotals by quarter. This is outputted in a neat table.

**Links:** This file heavily uses functions defined in GetPoints.php, which is include()’d at the top of the file.

**Methodology:**

* Retrieve the year from the previous page. This input is passed through standard sanitization procedure in case of hilariously bad typos or deliberate manipulation, neither of which is expected. Also define a variable for the second calendar year of the school year.
* Connect to MySQL
* Query MySQL to get a distinct list of NetIDs (ie, a list with no repeats)
* Delete the old points summary table. This table is regenerated each time this page is run; it serves only as simple means of organizing the data, as well as allowing additional export options through phpMyAdmin
* Run a short loop to pull the list of NetIDs into a php array.
* Use the function GetQuarterDates() to get the 4 critical quarter begin/end dates for the inputted year. Then use the standard php function strtotime() to reformat the dates properly.
* For each NetID, check first if any record is found over the entire year. If so, retrieve the name and points per quarter using the functions GetName() and GetPointsByPeriod(). Also count a total. Insert this data into the points summary table.
* Sort the points summary table, then loop again to output an HTML table with all the information.

### CheckPoints.php

**Purpose:** This is a lead-in page to check a single point summary by NetID. On this page, a user enters a NetID; on the next page, the record is retrieved and shown.

**Methodology:** Just a simple html form. NetID may be remembered and automatically suggested if the user has previously submitted points during the same session.

### CheckPoints2.php

**Purpose:** This page displays a summary of all the points subtotals by quarter based on the NetID supplied on the previous page.

**Caveats:** Unlike AdminTotals.php, the date is forced to be the current year, in the interest of simplicity. Also, a friendly reminder that this is not secure: anyone can check a points record, as long as they know someone’s NetID.

**Links:** This file heavily uses functions defined in GetPoints.php, which is include()’d at the top of the file.

**Methodology:**

* Fetch the NetID, running it through standard sanitization techniques. Also note that the NetID is forced lowercase.
* Get the name associated with the NetID with GetName(). Note that this function also detects NetIDs with no records associated.
* If there are records, choose the year based on the current date, get the quarter dates associated with that year using GetQuarterDates(). Format these dates as necessary.
* Use GetPointsByPeriod(). This function gives a lot of information, including the sum totals of all approved, rejected, pending, and restricted points. Output the following, on a per-quarter basis:
  + Total points
  + Number of total points that are also restricted
  + Number of points pending (not shown if 0)
  + Number of points rejected (not shown if 0)
* Also show a grand total for the year
* Provide a link back to the default points page

### default.php

**Purpose:** The “main page” for the points site. Just two buttons to check or submit points, plus the chapin crest.

**Methodology:** The buttons are each a separate form, with the action set to the page they point to. Also, a few more links are given. The CSS is inelegant here, but doesn’t have to be nice, since there’s only one image on the entire site.

### footer.php

**Purpose:** Like header.php, this file is appended onto every other file on the site, using the php include() function. Currently the only purpose of the footer is to provide my copyright.

**Note:** The copyright is my subtle, polite way of reminding everyone that I spent a month of my summer programming this website *pro bono*. Which I’m perfectly OK with (I’m an engineer, after all. This is interesting and fun for me), but I politely request you leave credit where credit is due.

**Legal:** The Chapin Residential College has my full, unconditional, and permanent license to the source code and all other associated resources within this site. I request, but do not assert, that the copyright statement be retained as credit for original authorship.

### FormP1.php

**Purpose:** When a user clicks “submit a points record,” they end up here. This first part of the form records name, NetID, and points category

**Methodology:** A few links, followed by an HTML form; fairly standard. Note that the $\_SESSION variable is later used to store name and NetID; if one is found, it will be entered in as the suggested value for the respective field, to facilitate entering multiple records quickly.

### FormP2.php

**Purpose:** After submitting name, NetID, and category, this page generates an appropriate list of events for the selected category. Additional fields may be generated if necessary (such as a “hours of service” or “points requested” field), as well as info and date fields.

**Methodology:**

* Read in the Name and NetID from $\_POST, running them through standard input sanitization. Also note that NetID is forced lowercase.
* Store Name, NetID, and Category in $\_SESSION, as they will be used both on return to P1 and to submit data all at once on P3.
* Echo back the data received, in roughly the same position as it was onscreen as a form. Now, however, it is un-editable text. (a back link is provided at the top in case errors are spotted)
* Create a form, then a massive switch statement
* For each possible points category, create a select form element with all events under said category. Also create a Description field (which can vary in size and title depending on the category), and possibly a “points” input, which takes a number of variable meaning (either points or hours of service).
* Also in the switch, note that every event is given a unique, easy to remember code, composed of the first two letters of the category, followed by a number. This is the value that is sent to the next page when the option is selected.
* After the switch, add two more elements (common to all points categories): A Date field and a submit button. The date field has an ID that singles it out for a JavaScript function to make it a nice “datepicker” field rather than a simple text field.

### FormP3.php

**Purpose:** This page assembles are the info collected on the previous two, and interprets everything, before submitting a nice, neat entry into the raw submissions table. Then it echos back a short confirmation and links to submit a new record or return to home.

**Methodology:**

* Retrieve values stored in $\_SESSION (name, netid, category, from P1) and in $\_POST (event, date, points)
* Points is probably blank; it’s only used when the number of points isn’t set or when hours of service is a relevant field (even if the “points” field is available, it will still be ignored below if it’s not necessary). Set points to 0, if it is blank, otherwise read the value.
* Now, another gigantic switch statement. Here we switch on the $event variable, which is the same 3-charcter alphanumeric code assigned on P2\*\*. For each code, there is a corresponding number of points (assigned to $points), and an event name, which is saved back into the $event variable. Here we may also set $restricted to TRUE if necessary ($restricted is set to FALSE at the top of the switch, so there’s no need to put any statement for unrestricted events).
  + Note that any variable points or variable hours of service are here rounded and saved back into $points. For hours of service, the value stored in $points is multiplied by two, since 1 hour of service (has an initial value of $points=1) is worth two points.
  + The switch statement is organized by category, but the order and organization has no functional effect
* A final catch to get rid of any negative points
* Connect to MySQL
* Insert the record. Now that all the data has been scrubbed clean, this is very easy to do
* Echo back some basic information about the points submitted, as well as general confirmation.

**\*\*Side Note:** The code system is inelegant, but it works. The obvious alternative would have been to assign each option a plaintext name, like <option value=”Attending Fireside”> etc. However, this exact text string needs to be copied directly into FormP3; any small typos, changes in phrasing, or changes in capitalization would have thrown off entries and caused matching problems. Plus, special characters would have been tricky (think: Master**’**s table). So I went with the confusing code system, ironically because I thought it would be easier to maintain. I could be completely wrong, but my general coding instinct tells me I’m not. Probably.

### GetPoints.php

**Purpose:** This is a function file. This, unlike most files, does not contain HTML, and should never be intentionally read by a browser. Instead, this is include()’d where necessary, and provides functions for accessing points, dates, and names from MySQL. The goal of this file was to provide functions useful both for checking individual records and for generating a master list.

**Methodology:**  
*(This section split by function name. Note that the include() statement at the top affects all 3 functions here)*

*GetQuarterDates($year)*

* Connect to MySQL
* Create a query to retrieve 4 dates based on the year supplied to the function
* Save the entire row as $date (This has fields $date[0]…$date[3])
* Return $date.

*GetPointsByPeriod($netid, $startdate, $enddate)*

* Connect to mysql
* The goal here is to create 4 queries, with code to processes the returned results. These sections are as follows
  + Query the sum of all approved, unrestricted points within the date range for the $netid provided. Sum them and store the value as $unrestricted
  + Query the sum of all approved, restricted points within the date range for the $netid provided. Sum them and store the value as $restricted. Cap this at a maximum of 20, if it is over.
  + Query the sum of all pending (not yet approved or rejected) points within the date range for the $netid provided. Sum them and store the value as $pending.
  + Query the sum of all rejected points within the date range for the $netid provided. Sum them and store the value as $rejected.
* In addition, the total valid points ($restricted+$unrestricted) is calculated
* Return $points, which has 5 fields: restricted, unrestricted, total, pending, and rejected.

*GetName($netid, $startdate, $enddate)*

* The syntax here with $startdate and $enddate set to NULL means that they will take no value, unless explicitly provided.
* If no $startdate and $enddate are provided, query the entire table for entries of $netid. We order by name supplied with the given NetID, and select the most common name. This allows some flexibility with one’s “name” – a typo, change in name or spelling shouldn’t cause problems. We just pick the most common self-supplied name.\*\*
* Return the name. If no single record is found for the NetID, return the name as ‘INVALID\_NETID,’ which means that no records were found, and thus the NetID is wrong (or just belongs to a slacker chapino).
* If dates are provided, do the same, but only check NetIDs in the date range. Because the two forms of this function are both called in various places, this means that a NetID could easily show up as existing, but invalid because of the date restriction. Generally, the results of this are seen when a NetID shows up on a list as having 0 points: It’s valid, but out of range.
* Return the name.

**\*\*Side Note:** In the interest of security, it is perhaps slightly more secure to require that your name always match exactly your previous entry, so my must have the proper name/NetID pair to access a point summary. But this would cause problems with typos, name changes, etc, and I trust Chapinos enough to not have to institute anything past the barest security possible.

### Header.php

**Purpose:** like footer.php, this file is appended to each file on the site, but at the top this time. This provides many functions, scripts, and other crucial information needed by most pages. Specifically, this file provides all other files with access to:

* An open <html> tag. This is important: a file with header.php does not need an <html> tag, but **does** need an </html> tag.
* A populated header including:
  + A meta tag that sets some basic information and flags to prevent this from showing up on Google
  + A title (Shown on the tab of your browser, among other places)
  + Links to stylesheets (see styles.css) and jquery code (I know no JavaScript. I cannot emphasize this enough. These scripts were copied from open forums found on google, and will probably still work in a few years. Some of the stylesheets came with this)
  + 4 datepicker scripts. Again, inelegant just-barely-works JavaScript
* Outside the head, there is a section of php. Here we define
  + A function for connecting to MySQL, which includes all the login information
  + A function for sanitizing inputs, to prevent (accidental or intentional) SQL injection
  + The standard connection variables, outside the function and thus available independently to all the code.

### Points system manual.docx

**Purpose:** Recursion

### Styles.css

**Purpose:** CSS stylesheet. This controls the “look and feel” of the website. If I did my job right (spoiler: I probably didn’t), small changes to this should be able to drastically update the look of the entire site.

**Specific, used styles:**

* Body – All this does is set the background color to the page
* H1 – Sets all page headers
* Form – this was ignored. I’m not sure what would happen if you started tweaking the form CSS, and I’m not eager to try
* #container – This sets the friendly, inviting box at the center of the page.
* #copy, #copy.a – Sets the copyright at the bottom of the page (I think)
* Img – Every image on the site is floating to the right at 50px padding. There is only one image on the site. If there are ever two images on this site, there could be a problem
* .wide – sets a wider version of #container. Used on AdminApproval.php to accommodate the very large table created
* .error – sets red error text, used for the asterisk on required fields.
* Table, td – sets table body formatting
* Th – sets table header formatting

# Appendix A: 10 minute Crash-Courses in HTML/Javascript/CSS/PHP/SQL

We’ll be using the “fake it ‘till you make it” philosophy, here

I’ll preface this section by saying that Google is your friend. A little knowledge of syntax, however, can go a long way to understanding what you’re reading, so what follows is a brief description of how these languages are structured and what they’re used for.

An excellent resource for most of these is <http://www.w3schools.com/>; I use this site as a reference manual for various tags or functions, and their “try it” editor is very good for showing examples and letting you fiddle with them and be able to see the immediate results. If you’re looking to do something basic, this is a good place to find a simple example to tweak.

### HTML

**What’s it for:** HTML codes for how a web page is displayed. Any words that actually appear onscreen will be written in HTML. In addition to that raw content, HTML consists of a system of tags which modify the content they are associated with; this is how headers, for example, are formatted differently than ordinary paragraphs. Furthermore, more complex tags may have additional attributes set within the tag that further modify the behavior.

**How is it interpreted:** HTML is read by the browser. Whenever you open a webpage, your computer is sent the raw HTML, which is then interpreted by the browser; the result is what you see on screen. Any .htm or .html file can be opened and viewed directly by a web browser. If you want to see and edit the actual tags, you can open it in notepad or Notepad++. If you’d like to view the HTML of any webpage, most browsers offer a “view source” option to see the raw HTML.

Other file types may contain HTML. In particular, a .php file will generally contain html, and can even be primarily or entirely HTML.

**Syntax:** HTML is very readable, even without computer science experience. Tags are the fundamental unit of HTML code; tags generally come in pairs, with raw text sandwiched between them like this:

<Tag>text to display</tag>

As you can see, tags are enclosed in <brackets>, and a tag containing a /backslash signify the end of a tagged statement. Tagged statements can be nested, like such:

<p> a bit of this text is <b> bold,</b> but not all of it </p>

Unlike the first example, this is actual HTML. <p> is the tag for “paragraph,” so all of this text is formatted by the browser as a paragraph. The word “bold” is enclosed in an additional tag, <b>, which signifies “bold.” So this would actually be seen by the browser as

a bit of this text is **bold,** but not all of it

Other common formatting tags include <i> for italics and <h1> through <h6> for different levels of headers (The displayed title of the page is generally in <h1>). The special tag <br> can be used to create a line break (generally, ending a paragraph and beginning a new one has the same effect). Since <br> doesn’t have a corresponding end tag, it is sometimes written as <br /> (making it it’s own end tag), but this is not necessary.

**Structure:** An HTML document is generally structured as follows:

<html>

<head>

<!-- Header information: linked documents and stylesheets, titles, scripts, and other non-displayed page attributes -->

</head>

<body>

<!-- Body information: paragraphs, forms, tables, etc -->

</body>

</html>

Note that all html documents begin with the <html> tag and end with the corresponding end tag. Most of the webpage’s actual content is within the body, while most of the other information associated with it is in the head. Comments (text that is ignored by the browser) are written between <!-- and -->

**Important examples:** The most important use of HTML in this site is for generating forms. All the text boxes, dropdown menus, and submit buttons are coded in HTML. For that reason, it’s good to know the basic form of… well, a form.

<form action="FormP2.php" method="post">

Name: <input type="text" name="name" required> <br />

NetID: <input type="text" name="netid" required> <br />

<select name="category">

<option value="Academic">Academic Events</option>

<option value="Communications">Communications Events

</option>

</select>

<input type="submit" name='submit' value="Continue">

</form>

This is a simplified version of the code that runs the first form element of the points submission page. Nesting of tags is shown by indentation. The exact code can be found in FormP1.php.

The first thing you should notice is that these tags have attributes. So rather than a simple <form> tag, now the form is given two attributes: “action,” which determines what script or page should be called when the form is submitted, and “method,” which determines how the data collected is sent. (in all of my code, “method” will always be “post.” The other option is “get,” which is less ideal for this application). Some other attributes to pay attention to:

* **Input Type:** Determines how an input functions. “submit” makes it a button, “text” makes it a text box
* **Input Name/Select name:** These set the name under which data is stored; it is under this name that your php handling script should access the data submitted.
* **Option value:** This is the value that is stored under the <select>’s name when this option is selected.
* **Value:** This is what is displayed on a button or the default, displayed value in a text field.

Note that any text not enclosed in <> brackets is generally displayed somewhere on the page. So the text “NetID:” is displayed in front of a text field named “netid,” and the dropdown element has two options that appear as “Academic Events” or “Communications Events”

### CSS

This is the one I most expect someone in Chapin to have a passing expertise in. Ask your resident graphic designer (I’m sure you have one, and I’m sure they’ve at least seen this before)

**What’s it for:** HTML was created in the days where webpages were little blocks of black text on white background used for exchanging info between universities and military bases. When things started getting a bit more democratized, someone starting adding formatting to HTML, but it was quickly apparent that that was a terrible idea, since some poor shmuck would have to update every single HTML document to make a small style change to a site. Thus CSS was born

**How it is interpreted:** Just like HTML, CSS is read by the browser. It basically tells your browser what the average header, image, or paragraph should look like, or at least how it should be formatted

**Structure:** CSS is basically at its heart just a list. An element, like p (<p> is html for paragraph) or body or form is stated, followed by its default formatting in {braces}. These formattings are in the form of

Formatting: parameter;

So headers might have a color: attribute (with a color stated either as a predefined color like ‘black’ or ‘BlanchedAlmond’, or as a hex/RGM color code, like the corresponding #000000 or #FFEBCD [there’s a lot of weird CSS colors. I highly recommend reading the full list]). There are also various font attributes, as well as things like padding, margin, and float, which control position. There may also be multiple parameters for a given attribute.

**Use:** If you really want to spruce up the site, you can learn a bit of CSS to change the appearance. Otherwise, this is very non-critical to the function of the site. See also Styles.css under the full documentation for specifics of how this was used for this particular site.

### JavaScript

I’ll be straight on this one: After learning PHP and SQL, plus re-learning HTML (I last touched it in 7th grade, I think), I wasn’t eager to learn another language. So I didn’t write any of the JavaScript in these files; I did however use a bit of code I pulled for a few things, most notably for making a “datepicker” widget to make selecting dates user-friendly and easy. So I don’t have a lot to say on this one. Still, a few points:

**What it’s for:** HTML is good for defining static content. But when you want a webpage to interact with a user, HTML is very limited, beyond a few form elements that have pre-defined behavior. What if you want to make the title purple whenever the mouse hovers over it? Or, more practically, make form elements fill themselves in based on previous answers, all in real-time. JavaScript runs in real-time, and is perfect for these sort of interactions.

**How it is interpreted:** Again, JavaScript is interpreted by the browser. This means it’s good at figuring out what the user is doing (where the mouse is, what text is being typed, etc), but not very good at communicating with the outside world; it’s basically “stuck” inside your browser, like HTML.

**Structure:** All you really need to know (read, “all I actually know”) is that JavaScript can be written within a <script></script> pair of tags within the header. You’re on your own for the rest on this language. My very inelegant and clumsy syntax should attest to that.

**What I should have used it for:** All of the forms on this site are on multiple pages: at least one to enter data and another to view the result. JavaScript would be a useful tool for making the form respond dynamically. This would have been especially useful on the main submission form, where you have to enter a category, then proceed to the next page to see the events available for that category. With javascript helping out, the form could have been on a single page, and could have dynamically changed itself if you switched to a different category midway through filling it out.

### PHP

**What’s it for:** PHP is a full, functional programming language specifically for use in web-based applications. It can be used to create programming loops, branching, and other complex logic, along with any other programming staple such variables, functions, and even classes/objects. In addition to that, PHP has robust tools for interacting with databases and html forms, making it ideal for most of the heavy lifting of this website. If you have ever programmed in C, C++, Java, or Python, PHP should feel very familiar; in particular, the syntax is nearly identical to C.

**How it is interpreted:** Unlike HTML and JavaScript, PHP is interpreted by the webhost. The raw php code never reaches the user’s web browser; instead, the webhost reads the PHP, interprets it, and assembles HTML code from it to send to the browser. Because PHP is used to assemble HTML, it generally has a lot of HTML (and possibly JavaScript) embedded within it.

Because PHP is interpreted by the webhost, it only runs when the webhost sends or receives data. This translates to it running only when a page is loaded, or when a form having a PHP handling script is submitted. It does **not** run in real time or dynamically update, which limits the interactivity of PHP to a page-to-page level.

**Syntax:** Anything after //two backslashes in a line is commented and ignored during run. Every statement in PHP is terminated by a semicolon. Variables always begin with a $dollarsign followed by an underscore or a letter (variable names can include numbers, but cannot begin with one). PHP code always is bookended by <?php and ?> . Everything outside of those brackets is interpreted as HTML code.

**Structure:** Let’s look at some elementary form handling. In the previous HTML example, I showed a simplified form to submit name, NetID, and a points category. Here’s a simplified version of part of the handling code for that, excepted and adapted from “FormP2.php”:

<?php

// get the 3 variables requested

$name = sanitize\_input($\_POST['name']);

$netid = strtolower(sanitize\_input($\_POST['netid']));

$category= sanitize\_input($\_POST['category']);

echo "Name: $name <br /><br />

NetID: $netid <br /><br />

Category: $category <br /><br />";

?>

As you can see, there’s a lot going on here. Because this is form handling code for a form with “method=post” (remember that?), variables imputed into the form are stored in the $\_POST variable. $\_POST is actually an array – it has several different values associated with it, each stored in a separate “field,” which is referenced by whatever is in [brackets] after it. The field can be a numerical index, or, in this case, it can be named. So $\_POST[‘name’] has the value of what the user entered as their name, $\_POST[‘netid’] as the value of their netid, etc. How do I know the name of these fields? That was determined by the ‘name=”name” ‘ or ‘name=”netid” ‘ attribute back in our HTML example.

In a perfect world, the user would always input their name and netid in exactly the format I want. But since that’s not a case, I can’t just store $\_POST[‘name’] as the person’s name – that could cause problems. So I call a **function** on it: sanitize\_input(). This function is defined in header.php and, because this file is included (using the include() function, of course) on this file, the function can be called here. Inside the parentheses I pass the function an argument ($\_POST[‘name’]); the function operates on that argument and then takes on a new value (the ‘sanitized’ text, which is now missing a few troublesome special charcters), which is now safely stored as $name. The same is done for $netid, but an additional function, strtolower(), is called on the sanitized input. This function is predefined in PHP, and it simply converts any uppercase letters to lowercase. This is important to prevent duplicate records.

Finally, the “echo” command displays the name , NetID, and category received back on the page. Notice how the echo command is perfectly capable of echoing HTML tags – these will be interpreted by the browser like normal HTML code.

Oh, and one other thing: ‘single quotes’ and “double quotes” aren’t quite the same in PHP. In general, double quotes will allow interpretation of what’s inside, so $name shows up as the person’s name, not the literal text “$name.” Similarly, double quotes allow for the echoing of HTML. Single quotes are less permissive and generally will not pass along any formatting, so variable names and html tags will be rendered literally and not interpreted.

**Important Example:** Let’s take a look at the code that comes right after the above example:

?>

<form action="FormP3.php" method="post">

<?php

switch($category) {

case 'Academic':

?>

Event: <select name="event">

...

At the top, we break the PHP code to run a line of HTML, and then start up another section of PHP. As mentioned before, PHP files consist of a lot of HTML, so this shouldn’t be too surprising.

Next we move into a switch. The switch statement is very common in this site: it create a long conditional. (If you know some basic programming, it’s basically a long chain of if()…elseif()…elseif() etc) Here, it takes the value of $category, and compares it to each case statement. If there’s a match, it will run all code below the case statement, until hitting a break, which terminates the switch (code execution next moves to whatever is after the ‘}’ which ends the switch statement).

At this point, you may notice something weird: I start a case statement, then abruptly end the php code without properly ending the case or the switch, and start up in HTML! While this seems like this should break something, this is actually valid syntax. Because the PHP is still being interpreted, the HTML within the case statement *has now become conditional*: That means that the “Event: <select name=”event”>” and all html that follows all the way until “<?php break;” *will only be read if the case statement matches $category*.

This is a very powerful technique for mixing HTML elements with PHP functionality. However, it tends to be a bit of a brain\*\*\*\* to read and comprehend. In some other cases, the code might just echo the HTML rather than breaking the PHP and restarting it, which perhaps makes a little more sense (though is harder to actually read) but the effect is the same. Oh, well; nothing in life is perfect.

**Special Functions**: For the purpose of a 10 minute guide, I can’t explain everything you need to know to write useful PHP, but I can at least point you in the right direction. So here’s a few functions you should be aware of, for use in accessing an utilizing MySQL:

* connect\_to\_mySQL() – This is defined in header.php (which is include()ed in basically every file on the site). This function establishes a connection with the points database, and should save you the trouble of remembering a few specific connection function and error handlers. Call this before you do anything that accesses MySQL. This optionally returns a variable that references the connection, if you need that information.
* mysql\_query($sql) – This runs a query (named here $sql), and returns a results variable. For how to create and use queries, read up on SQL below.
* mysql\_fetch\_row($result) – If your query is expected to return a table, use this to read one row of it. The result is stored in an array which can be numerically indexed starting at 0. This can be run in a loop and return the next row each time, generally as “while($row = mysql\_fetch\_row($result))”; such a loop will yield the entire table, one row at a time
* mysql\_fetch\_assoc($result) – If you’re expecting a single value to be returned by a query, this should be used instead of fetching a row.

### SQL

SQL stands for “structured query language.” Unlike HTML (“hypertext markup language”) or PHP (“I still don’t know what this stands for”), this is an acronym worth knowing, because SQL does exactly what is says on the tin.

**What it’s used for:** Databases hold lots of useful information. For the most part, this information is stored neatly in organized tables. But when you want to sort through that data in any useful way, you’ll need a way to query the database to get what you need. That’s what SQL is used for: It’s a structured way to query a database (Thus the name.)

**How it is interpreted:** A database will be sent an SQL query, which it will interpret, run, and return the results of. In this way, it is similar to PHP, and it will be run whenever PHP code invoking it is run. However, it’s important to remember that the database is an entity separate from the webhost, so the SQL is not being run in the same “place” as the PHP is.

For this site, our database is MySQL. Access to this is provided by the webhost. So there’s a subtle distinction.

**Syntax:** SQL is very easy to read, as it mimics plaintext and generally does what you’d expect the command to do just by reading it. Commonly, commands and keywords are written in all caps, though this is not strictly necessary. Although these keywords are not case-sensitive, table names, field values, and so forth may be, so tread with care.

**Examples:** There are two main functions you should know in SQL: Writing a record to a table, and finding and returning a record from a table. Let’s look first at adding a record, here from “FormP3.php”.

$sql = "INSERT INTO Raw\_Submissions (Name, NetID, Category, Event, Date, Additional\_Info, Points, Restricted)

VALUES ('$name', '$netid', '$category', '$event', '$date', '$info', '$points', '$restricted');";

The keyword “INSERT INTO” tells the database we want to add a record. This is followed by the table name (“Raw\_Submissions”), and then in parentheses, a list of table headings we’ll be modifying. Below that is the keyword “VALUES”, followed by the values we are adding, in the same order as the headings listed above.

Note that this is actually a code snippet in PHP. Everything between the two “double quotes” is our sql query; everything outside is purely PHP. Also, note that because the query is in double quotes, the variable names $name, $netid, etc will be replaced with their actual values at runtime.

Now, a retrieval example, from “AdminApproval.php”:

$sql = 'SELECT \* FROM Raw\_Submissions

WHERE Approval\_Status IS NULL;';

Here, there’s no variable names to be interpreted, so simple ‘single quotes’ are used. You may have noticed in the previous example that “approval\_status,” although a valid column header in “Raw\_Submissions,” is not set when a record is submitted. It is therefore “null” (it has no value). Here, we’re merely selecting all records (\* means “all” or “all columns”) from the table for which the Approval status hasn’t been set.

A few other SQL techniques are used in the code, especially in “GetPoints.php”; This would be another good place to look for sample SQL snippets.