Table of Contents

[Introduction (If nothing else, read this) 2](#_Toc427416457)

[Things You’ll Need 3](#_Toc427416458)

[What this system will and will not do 4](#_Toc427416459)

[How it works, in a nutshell 4](#_Toc427416460)

[Limitations 4](#_Toc427416461)

[The basics of managing points 6](#_Toc427416462)

[The Responsibilities of the VP: “What you have to do” 6](#_Toc427416463)

[phpMyAdmin 6](#_Toc427416464)

[Modifying the points system 8](#_Toc427416465)

[The basics 8](#_Toc427416466)

[Changing Point Values or Restricted Status 9](#_Toc427416467)

[Adding a new event 10](#_Toc427416468)

[Getting Advanced: Adding variable points 11](#_Toc427416469)

[Final Exam: Adding an entire points category 12](#_Toc427416470)

[But What about …? 13](#_Toc427416471)

[Full Documentation 14](#_Toc427416472)

[Appendix A: 5 minute Crash-Courses in HTML/Javascript/CSS/PHP/SQL 15](#_Toc427416473)

# 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 organized so that you can most easily find what you need; 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 some other coalition within 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 read up on 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  
8/15/15

# 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), 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 begin bolding 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 records. 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.” So typo’ed names (or weird capitalizations, middle names, creative spellings) are not a big problem.

### 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 robust tools for editing or viewing specific submissions. The phpMyAdmin tool is available for viewing the raw database, and allows the use of SQL to perform more advanced operations and sortings, but there is no interface for these sorts of operations within the website itself. 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’s 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 has no control over for which date records are shown. The master list, however (accessible only by the admin) can be created for different years.

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 or 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.

# 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)
2. **Generate a master list:** You’ll need this for housing selection, and possibly for determining voting rights. *Be sure to approve/reject all pending points first!*
3. **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.
4. **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. Login 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 containts (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**.

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. 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

### 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 the 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;

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;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 request, 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 record and the record for which the code is the same will both create the same record. Note also that this needs to 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 set. 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, but here, $points does have a 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 $points (*side note: in code, statements like “$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 Full Documentation for this file if you’re really curious.

### 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, but the 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.

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.

# Appendix A: 5 minute Crash-Courses in HTML/Javascript/CSS/PHP/SQL

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