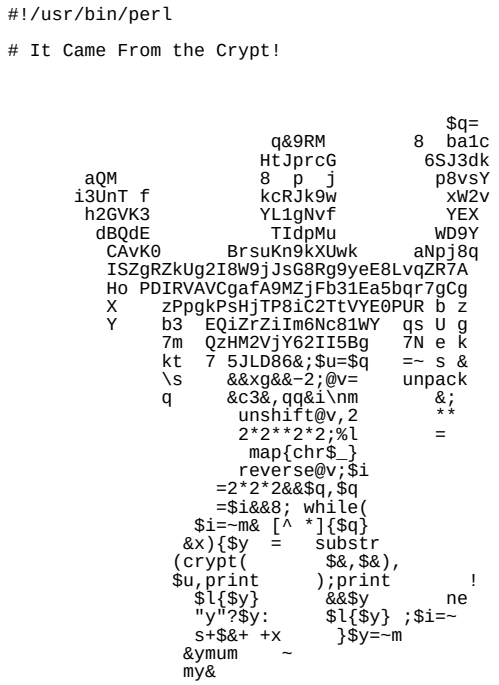
CGI Programming with Perl



Kirrily Robert
Paul Fenwick
Jacinta Richardson

CGI Programming with Perl

by Kirrily Robert, Paul Fenwick, and Jacinta Richardson

Open Publications License 1.0

This document is a greatly revised and edited copy of the training notes originally created by Kirrily Robert and Netizen Pty Ltd. These revisions were made by Paul Fenwick and Jacinta Richardson.
Copies of the original training manuals can be found at http://sourceforge.net/projets/spork
Copies of the Obsidian training manuals can be found at http://www.obsidian.com.au/training
This training manual is maintained by Perl Training Australia, and can be found at http://www.perltraining.com.au/notes.html
This is version 4.1 of Perl Training Australia's "Perl Programming with CGI" training manual.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

Welcome to Perl Training Australia's *CGI Programming with Perl* training course. This is a one-day course in which you will learn how to write dynamic, interactive web applications using the Perl programming language.

Course outline

- What is CGI?
- Generating web pages with a Perl script
- HTML forms and CGI.pm
- Accepting and processing form input
- Security issues
- Splitting HTML and code with HTML::Template

Assumed knowledge

It is assumed that you know and understand the following topics:

- Unix - logging in, creating and editing files
- Perl - variable types, operators and functions, conditional constructs, subroutines, basic regular expressions
- Basic HTML - paragraphs, headings, unordered lists, anchor tags, images, etc.

If you need help with editing files under Unix, a cheat-sheet is available in Appendix A and an editor command summary in Appendix B.

The Unix operating system commands you will need are mentioned and explained very briefly throughout the course - please feel free to ask if you need more help. The required Perl knowledge was covered in our "Introduction to Perl" course. If you've worked with Perl for about a year you'll know much of the required material. Lastly, an HTML cheat-sheet is provided in Appendix D for those who need reminding.

Module objectives

- Understand the meaning of CGI and the Hypertext Transfer Protocol
- Know how to generate simple web pages using Perl
- Understand how to accept and process data from web forms using the CGI module
- Understand security issues pertaining to CGI programming and how to avoid security problems
- Recognise and use a number of Perl modules for purposes related to CGI programming

Platform and version details

This module is taught using Unix or a Unix-like operating system. Most of what is learnt will work equally well on Windows NT or other operating systems; your instructor will inform you throughout the course of any areas which differ.

All Perl Training Australia's Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs significantly from previous versions of Perl, so you will need a Perl 5 interpreter to use what you have learnt. However, older Perl programs should work fine under Perl 5.

At the time of writing, the most recent stable release of Perl is version 5.6.1, however older versions of Perl 5 are still common. Your instructor will inform you of any features which may not exist in older versions.

The web server used for examples in this module is Apache (http://www.apache.org). We have chosen this web server for examples as it is freely available, widely used, and very fast and full-featured.

The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographic conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what appears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appears *`like this`*


Notes and tips appear offset from the text like this.


Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.


Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.


Notes marked "Caution" contain details of unexpected behaviour or traps for the unwary.

Other materials

In addition to these notes, it is highly recommend that you obtain a copy of Programming Perl (2nd or 3rd edition) by Larry Wall, et al., more commonly referred to as "the Camel book". While these notes have been developed to be useful in their own right, the Camel book covers an extensive range of topics not covered in this course, and discusses the concepts covered in these notes in much more detail. The Camel Book is considered to be the definitive reference book for the Perl programming language.

The page references in these notes refer to the *3rd edition* of the camel book. References to the 2nd edition will be shown in parentheses.

Chapter 2. What is CGI?

In this chapter...

In this section we will define the term CGI and learn how web servers use CGI to provide dynamic and interactive material. We explore the Hypertext Transfer Protocol as it applies to both static and CGI-generated content, and examine raw HTTP requests and responses by telnetting to a web server.

Definition of CGI

CGI is the *Common Gateway Interface*, a standard for programs to interface with information servers such as HTTP (web) servers. CGI allows the HTTP server to run an executable program or script in response to a user request, and generate output on the fly. This allows web developers to create dynamic and interactive web pages.

CGI programs can be written in any language. Perl is a very common language for CGI programming as it is largely platform independent and the language's features make it very easy to write powerful applications. However, some CGI programs are written in C, shell script, or other languages.

It is important to remember that CGI is not a language in itself. CGI is merely a type of program which can be written in any language.

Introduction to HTTP

To understand how CGI works, you need some understanding of how HTTP works.

HTTP stands for HyperText Transfer Protocol, and (not very surprisingly) is the protocol used for transferring hypertext documents such as HTML pages on the World Wide Web.

For the purposes of this course, we will only be looking at HTTP version 1.0. The current version, 1.1, is specified in RFC 2068 and contains many more features, but none of them are necessary for a basic understanding of CGI programming. An HTTP cheat-sheet, containing some common terminology and a table of status codes, appears in Appendix E.


RFCs, or "Request For Comment" documents, can be obtained from the Internet Engineering Task Force (IETF) website (http://www.ietf.org/) or from mirrors such as The RFC mirror at AARNet (http://mirror.aarnet.edu.au/pub/rfc).

Figure 2-1. A typical HTTP and CGI connection

A simple HTTP transaction, such as a request for a static HTML page, works as follows:

1. The user types a URL into his or her browser, or specifies a web address by some other means such as clicking on a link, choosing a bookmark, etc
2. The user agent connects to port upon which the HTTP server is running (usually port 80)
3. The user agent sends a request such as `GET /index.html`
4. The user agent may also send other headers
5. The HTTP server receives the request and finds the requested file in its filesystem
6. The HTTP server sends back some HTTP headers, followed by the contents of the requested file
7. The HTTP server closes the connection

When a user requests a CGI program, however, the process changes slightly:

1. The user agent sends a request as above
2. The HTTP server receives the request as above
3. The HTTP server finds the requested CGI program in its file system
4. The HTTP server executes the program
5. The program produces output, including HTTP headers
6. The HTTP server sends back the output of the program
7. The HTTP server closes the connection

Terminology

During this course we'll use a number of terms that have very specific meanings. A list of these terms can be found in Appendix E.

HTTP Methods

GET

The GET method means retrieve whatever information is identified by the request URI. If the request URI refers to a data-producing process (eg a CGI program), it is the produced data which is returned, and not the source text of the process.

HEAD

The HEAD method is identical to GET except that the server will only return the headers, not the body of the resource. The meta-information contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request. This method can be used to obtain meta-information about the resource without transferring the body itself.

POST

The POST method is used to request that the server use the information encoded in the request URI and use it to modify a resource such as:

- Annotation of an existing resource
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
- Providing data {such as the result of submitting a form} to a data-handling process
- Updating a database

What is needed to run Perl CGI programs?

There are several things you need in order to create and run Perl CGI programs.

- a web server
- web server configuration which gives you permission to run CGI
- a Perl interpreter
- appropriate Perl modules, such as CGI.pm
- a shell account is extremely useful but not essential

Most of the above requirements will need your system administrator or ISP to set them up for you. Some will be wary of allowing users to run CGI programs, and may require you to obey certain security regulations or pay extra for the privilege.

Chapter summary

- CGI stands for Common Gateway Interface
- HTTP stands for Hypertext Transfer Protocol. This is the protocol used for transferring documents and other files via the World Wide Web.
- HTTP clients (web browsers) send requests to HTTP (web) servers, which are answered with HTTP responses
- The request/response can be examined by telnetting to the appropriate port of a web server and typing in requests by hand.

Chapter 3. Generating web pages with Perl

In this chapter...

In this section, we will create a simple "Hello world" CGI program and run it, then extend upon that to integrate parts of Perl taught in previous modules. Alternative quoting mechanisms are briefly covered, and we also discuss debugging techniques for CGI programs.

Your public_html directory

The training server has been set up so that each user has their own web space underneath their home directory. All files which will be accessible via the web should be placed in the directory named `public_html`. This is common for most personal homepages.

The directory `~username/public_html` on the Unix file system maps to the URL `http://`*`hostname`*`/~`*`username`*`/` via the web.

Replace *`username`* with your username. Your instructor will tell you what the *`hostname`* is.

The CGI directory

CGI scripts are usually kept in a separate directory from plain HTML files. This directory is most commonly called `cgi-bin` (the "bin" stands for "binary" but really just means "executable files", whether compiled binaries or interpreted scripts such as Perl programs). The web server is usually set up so that you only have permission to run CGI programs from the `cgi-bin` directory, for security reasons.

Before proceeding any further, do the following:

1. Change to your `public_html` directory
2. If you type **ls** to get a directory listing, you will see that you have several HTML files here, as well as a `cgi-bin` directory.
3. Change to your `cgi-bin` directory and type **ls**, and you will see that the example scripts for this course are already installed here.

If you were setting this up for yourself, you would need to be sure of the following:

1. That your home directory is world executable
2. That your `public_html` directory is world executable
3. *That all your HTML files are world readable*
4. That your `cgi-bin` directory is world executable - note that it is not compulsory to have a `cgi-bin` directory - some server configurations allow you to execute a CGI script from any directory. Some servers won't let you execute CGI scripts at all.
5. **That all your CGI scripts are world readable and executable**

CGI.pm. Making CGI programming a breeze.

The CGI.pm library has been created to make a number of otherwise painful tasks easy. Using the CGI.pm library saves you from having to remember what HTTP headers look like, or which ones you have to set to stop your output being cached. It saves you from having to worry about how to generate or read cookies. It saves you from having to parse stuff to work out the users current URL. Most importantly saves you from having to worry about how to get to any the parameters passed to your script.

Not only does the CGI.pm library do all these nice things, it also gives you a functional interface to the HTML tags.

For those who've forgotten any HTML that they ever knew, don't start imagining that the code below is using Perl's `readline()` (`<>`) operator. That's just how HTML tags are written.

For example, instead of writing:

```
print "Content-type: text/html\n\n";
print qq{<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
Hello World
</BODY>
</HTML>};
```

You could write:

```
use CGI qw/:standard/;

print header,
      start_html({-title=>"Hello World",
                  -bgcolor=>"#FFFFFF"}),
      "Hello World",
      end_html;
```

Which, I'm sure you'll agree looks nicer, and saves you from having to remember all those HTML tags. (Later we'll talk about another module that you can use to remove almost all of of the HTML from your programs altogether).

That header thing

Now, we've jumped ahead a little here. What does that line

```
print "Content-type: text/html\n\n";
```

do for us? That's the HTTP header. Every CGI script must output a HTTP header giving a MIME content type, such as `Content-type: text/html`, with a blank line after it.

Most of you have probably realised that the line in the second example:

```
print header,
```

does the same thing. However, it's much easier to remember. The header function can also take arguments such as cookies and page expiry dates. We'll cover cookies soon.

If your output is of another MIME type, you should print out the appropriate `Content-type:` header - for instance, a CGI program which outputs a random GIF image would use `Content-type: image/gif`. Of course, since you'll be using CGI.pm you only need to write `print header('image/gif');`.

Quoting and roll-your-own quotes

For those unacquainted with `qq{}`, this is another form of quoting strings in Perl. `qq` stands for double quotes, the same idea works with `q` for single quotes and `qx` for backticks. Note that the same rules apply for each of these quoting styles as for their more common equivalents: `qq` allows variable interpolation whereas `q` does not. Quoting things with `qx` will send them to the shell for execution.

You may use any non-whitespace, non-alphanumeric character as your delimiters, pick the one least likely to appear in your string. Note that things that look like they should match up do. So `(` matches `)`, `{` matches `}`, < matches > etcetera. There are some illustrated below.

```
print qq(Jamie said "You should always watch your quotes!"\n);
print qq!Jamie said "These are Paul's favourite quotes". (He was wrong).\n!;
print qq[Jamie said "Perl programs ought to start with #!"\n];
print qq#Jamie said "My favourite regexp is '/[jamie]*/i;'"\n#;
```

Using CGI.pm should save you from having to worry too much about quoting text too often, but it's still bound to come up.

While we're here we'll briefly mention HERE documents. These can sometimes be confusing for the reader, and usually pick-your-own-quotes will be clearer and do a better job. We cover them here for the sake of completeness, and because they are still very common in older code.

HERE documents allow you to print text up unto a certain marking string. For example:

```
print <<"END";
I can print any text I want to put here without
fear of "weird" things happening to it.  All
punctuation etc is fine, unlike roll-your-own quotes,
where you have to pick some kind of punctuation to
delimit it.  Here, we just have to make sure that
the word, up there (next to print) does not appear
in this text, on a line by itself and unquoted.
Otherwise we terminate our text.
END
```

The quoting style used in HERE documents is whatever you quote the terminating word with next to the print (in this case double quotes). Use of double quotes guarantees variable interpolation, use of single quotes guarantees no variable interpolation and use of backticks passes each line as a command to the shell.

CGI.pm is a standard inclusion in all recent perl packages. Documentation for it can be found by reading **perldoc CGI**. It's also covered briefly in the Camel book on page 878 (but not in the 2nd Ed.).

Exercises

The point of these exercises is to get you used to various quoting styles. You'll be asked to do a few things with the CGI module, but for the moment, run your programs on the command line. When your script enters "offline mode", just hit **CTRL-d**.

1. Write a program which uses at least 7 different quote styles when printing out text.
2. Write a program which includes the CGI module and prints out a few lines of valid HTML (starts with a header, the start HTML tag, some other stuff and ends with the end HTML tag).

HTML output

As you've seen, all you need to do to generate HTML is to print it. (Making sure that you the first thing you ever print is always the header.)

The hello examples are already in your `cgi-bin` directory as `hello.cgi` and `hello2.cgi`.

HTML tags with CGI.pm

CGI.pm has functions for all the standard HTML tags and will pretend to have functions for any others you feel like creating. So, for example, if you want to print out an anchor:

```
<A HREF="http://www.perltraining.com.au/">Click here for further
information about Perl Training Australia.</A>
```

you'd type:

```
use CGI qw/:standard/;
print a({-href=>"http://www.perltraining.com.au/"},
        "Click here for further information about Perl Training Australia.");
```

If instead you wanted to print only the start tag, then generate some data, then print the end tag, you have to be a little more verbose.

```
use CGI qw/:standard start_ul end_ul/;
print start_ul;
foreach my $name (qw/Jacob Jeremy Jacinta Jenni Jack/)
{
        print li($name);
}
print end_ul;
```

The above is true for all tags excepting `<TR>`. If you try something like:

```
print table(tr(td()));  # This won't work.
```

Perl will assume that you mean to use the function `tr` which transliterates characters. In this case you have to use `Tr` instead.

```
print table(Tr(td()));  # This will work.
```

If you're happy with an object-oriented approach to CGI.pm You can change the list example above to:

```
use CGI;
my $cgi = CGI->new();
print $cgi->start_ul;
foreach my $name (qw/Jacob Jeremy Jacinta Jenni Jack/)
{
        print $cgi->li($name);
}
print $cgi->end_ul;
```

This saves you from having to worry about importing start and end tags explicitly for every single HTML function you might want to use (start_form, end_form, start_table, end_table...). It's also great for the purists out there who want complete control of their name space[1].

On the other hand, you have to make sure that the CGI object is available where ever you want to use its functions.

We use the functional notation throughout the course because the object-oriented style can make the code look very cluttered, and this can detract from its readability.

Faking tags with CGI.pm

In the previous section we mentioned that CGI would pretend to have functions for any tag you felt like creating. This is true. If you want to know how it works, ask your instructor in the next break.

This means that if the W3C announced a new tag: `<FISH>` (with it's close tag `</FISH>`) you can go straight into writing that into your CGI programs. You'd just write:

```
use CGI qw/:standard fish/;
print fish("My something here");
```

and everything would work fine (if your browser implemented fish tags).

If you use the CGI.pm module in an Object Oriented manner here, you do not need to import `fish` or any other functions. You're CGI object will be able to fake it for you anyway.

```
use CGI;
my $cgi = CGI->new();
print $cgi->fish("My something here");
```

Then again, why bother? (CGI D.W.I.M)

If you are still having doubts as to why a functional interface to HTML tags is a good thing, especially when you can just print out the HTML yourself, this section is for you.

Lets just imagine that you have the following code:

```
use CGI qw/:standard/;
print qq{<INPUT TYPE=TEXT NAME="name" VALUE="$name">};
print br;
print textfield({-name=>"name",
                 -value=>$name});
```

This should print out two input text fields, both called `name`, both initially containing the value of the `$name` variable. But lets just pretend for a moment that `$name` has quotation marks and a > character in it (amongst other things):

```
$name = qq{Simon ">\n<A HREF="http://www.some.bad.site">Click here!</A};
```

Then what is actually printed out is:

```
<INPUT TYPE=TEXT NAME="name" VALUE="Simon ">
<A HREF="http://www.some.bad.site">Click here!</A">
<BR>
<INPUT TYPE="TEXT" NAME="name" VALUE="Simon \"&gt;A
HREF="http://www.some.bad.site\"&gt;Click here!&lt;/A">
```

This is best seen by seeing how your browser interprets this. (This code is in `public_html/cgi-bin/whycgi.cgi`.)

What's happened is that CGI.pm has escaped the HTML in the variable for you, saving you from having to worry. Perl knows that this is what you mean. It can't do this for you in the literal HTML string because it does not know that what you are printing there is HTML. It's just a string.

By default CGI.pm will always escape HTML in the strings passed to any of its functions before printing them. Should you ever want to escape a string yourself CGI.pm provides a function called, intuitively enough `escapeHTML`.

CGI.pm has it's limits. If you do something like the following:

```
print start_h2;
foreach my $header (@headers)
{
        print $header, " ";
}
print end_h2;
```

the contents of each `$header` will not be escaped. This is because this is not called within a CGI.pm function. This is a good place to call `escapeHTML`.

Mind you, here would probably be a good time to do something like:

```
print h2("@headers");
```

which would escape the lot, but sometimes it's not that easy.

Running your CGI program

The simplest way to run your CGI program is to point your browser at it. In today's case you can see your copy of `hello.cgi` at `http://`*`hostname`*`/~`*`username`*`/cgi-bin/hello.cgi` where *`hostname`* is the hostname of the web server and *`username`* is your login name.

The most common mistake new CGI programmers make is to forget to set the permissions correctly on their programs. CGI scripts must be world readable and executable.

The second most common mistake is to print out something (anything) before printing out the header.

Try to watch out for these.

Exercises

Look forward to the next section if you encounter too many difficulties getting this to run

1. Look at the output of the `hello.cgi` and `hello2.cgi` programs with your browser.
2. Modify `hello2.cgi` to set a variable `$name` and include that name in the greeting.
3. Run your modified `hello2.cgi` script from the command line to ensure that it runs. (hit **CTRL-d** when your program pauses for input).
4. Now view your modified `hello2.cgi` script in your browser to see if your modifications worked correctly.

Debugging your CGI programs

When writing CGI programs, there are many problems which may affect their execution. Since these will not always be easily understood by examining the web browser output, there are other ways to determine what's going wrong.

If there seems to be a problem first try the following steps:

1. Check that your program compiles by using `perl -c`
2. Check the permissions on your cgi program. If it is not world executable then it won't work.
3. Run your program on the command line. When your program waits for input, this is your opportunity to pass in parameters. For the moment, press **CTRL-d**.
4. If your program runs fine on the command line but still does not output to the browser, make sure that you have not forgotten to print the header before any other output.
5. Check the HTML source that you're printing. Make sure that you've closed any tables you've opened and not made any obvious HTML errors.
6. Check the web server's log files. The location of these vary from system to system. On our system they're in `/var/log/apache/`.

Failing gracefully with CGI::Carp

The errors given in the web server's error logs are not always easy to read and understand. To make life easier, we can use a Perl module called `CGI::Carp` to add timestamps and other handy information to the logs.

```
use CGI::Carp;
```

We can also make our errors go to a separate log, by using the `carpout` part of the module. This needs to be done inside a `BEGIN` block in order to catch compiler errors as well as ones which occur at the interpretation stage.

```
BEGIN
{
        use CGI::Carp qw(carpout);
        open(LOG, ">>cgi-logs/mycgi-log") ||
                die("Unable to open mycgi-log: $!\n");
        carpout(\*LOG);
}
```

Lastly, and often the most useful use, we can cause any fatal errors to have their error messages and diagnostic information output directly to the browser:

```
use CGI::Carp 'fatalsToBrowser';
```


You can read all about CGI::Carp by checking out **perldoc CGI::Carp**. It's also covered briefly on page 878 of the Camel book (but not in 2nd Ed.).

Exercises

The following exercises practice using CGI to output different Perl data types such as arrays and hashes. Try using the CGI modules mentioned above. If you need help with HTML, there's a cheat sheet in Appendix D.

1. Write a CGI program which creates an array then outputs the items in an unordered list (HTML's `<ul>` element) using a foreach loop. This is mostly covered in an example above.
2. Modify your program to print out the keys and values in a hash, like this:
   - Name is Fred
   - Sex is male
   - Favourite colour is blue
3. Change your CGI program so that you output a table instead of an unordered list, with the keys in one column and the values in another. An example of how this could be done is in `cgi-bin/hashtable.cgi`

Cookies

We mentioned before that the `header` function can be used to send cookies to the user. We'll briefly mention how we do this here.

A cookie is a `(name,value)` pair. Your program may create one or more cookies and then send them to the browser, but only in the HTTP header. The browser will maintain a list of cookies and will return those that belong to a particular web server when requested.

As well as your `(name,value)` pair the cookie may have any of several optional attributes:

1. An expiration time
2. A domain
3. A path
4. A secure flag

To create a cookie we do the following:

```
use CGI qw/:standard/;
my $cookie = cookie({-name=>'userID',
                     -value=>'123456789',
                     -expires=>'+1h',           # 1 hours time
                     -path=>'/cgi-bin',
                     -domain=>'unimelb.edu.au',
                     -secure=>0});
print header({-cookie=>$cookie});
```

The name of the cookie can be any string at all. The value of the cookie can be any scalar value, including array and hash references. So:

```
my $cookie = cookie({-name=>'user preferences',
                     -value=>\%myhash});
```

is valid.

To create multiple cookies give the header function an array reference of cookies:

```
my $cookie1 = cookie({-name=>'user preferences',
                     -value=>\%myhash});
my $cookie2 = cookie({-name=>'userID',
                     -value=>'1234'});
print header({-cookie=>[$cookie1, $cookie2]});
```

To get the contents from a cookie, we request it by calling the `cookie` method with a name but no value:

```
use CGI qw/:standard/;

my $userID = cookie("userID");
my %user_preferences = cookie("user preferences");
```


Read the CGI documentation for all the ins and outs of cookies that we haven't covered here. **perldoc CGI**

Exercises

1. Write a program that creates a cookie and gives it to your browser (make sure that you have cookies turned on).
2. Modify your program to ask for that cookie and if it is present do something different.
3. Play with giving the browser different types of cookies (hash references, array references, scalars) and retrieving them, as well as giving the browser multiple cookies.

Environment variables

In Perl, there is a special variable called `%ENV` which contains all the environment variables which are set.

When a web server runs a CGI program, certain environment variables are set to provide information about the web server, the request made by the user agent, and other pertinent information.

Examples of environment variables available to your CGI script include HTTP_USER_AGENT which describes the user agent or browser used to make the request, and HTTP_REFERER, which indicates the referring page (if any).

Exercises

1. Modify your table-printing script from the previous exercise to print out the hash `%ENV`.
2. The `HTTP_USER_AGENT` environment variable contains the type of browser used to request the CGI script.
   - Write a script which prints out the user agent string for the requesting browser
   - Take a look at what various browsers report themselves as -- try Netscape, Internet Explorer, or Lynx from the Unix command line. You will note that Microsoft browsers purport to be "Mozilla compatible" (i.e. compatible with Netscape).
   - Use a regular expression to determine when a certain browser (for instance, Microsoft Internet Explorer) is being used, and output a message to the user.
3. The `HTTP_REFERER` (yes, it's spelt incorrectly in the protocol definition) environment variable contains the URL of the page from which the user followed a link to your CGI program. If you call up your CGI program by typing its URL straight into the browser, the `HTTP_REFERER` will be an empty string. Create a HTML page that points to your CGI program and see what the `REFERER` environment variable says.

Chapter summary

- CGI scripts are programs written in Perl or any other language that output web content such as HTML pages.
- The CGI.pm module, which comes standard with Perl distributions, provides a function based interface to HTML tags and headers.
- CGI scripts must output a Content-type header and a blank line before anything else. Using the CGI.pm module this can be done by `print header;`
- Perl quoting functions such as `qq{}` and `q{}` can be used to quote text without having to be concerned about double quotes and single quotes appearing in that text.
- Using CGI.pm HTML functions is as easy as printing the HTML tag you expect as a function, for example:

  ```
  print b("this is now bold");
  ```

  Excepting <TR> which must be called as

  ```
  print Tr(..);
  ```

- Running a CGI program is as easy as calling it from a web browser.
- Debugging techniques for CGI:
  - Check that the script compiles and runs from the command line.
  - Check that your script prints the header before any other output.
  - Check that your printed HTML is valid.
  - Check the logs

  Use CGI::Carp to make your life easier.

- The CGI.pm module makes creating and passing cookies very easy. Cookie values can be any kind of scalar. To get the values out of a cookie we call the cookie function without a value.
- The `%ENV` special variable can be used to access environment variables via CGI scripts, including such variables as HTTP_USER_AGENT and HTTP_REFERER

Notes

1. One of the authors of this course is such a name space purist after a particularly traumatic experience with large amounts of code written by someone who wasn't.

Chapter 4. HTML forms and CGI.pm

In this chapter...

This chapter focuses on generating HTML forms with CGI.pm.

A quick look at HTML forms

To be able to use CGI to accept user input, you will probably need to understand HTML forms. There's an HTML cheat-sheet in Appendix D of these notes, but here's a brief run-down of the major parts of HTML forms:

The FORM element

The `FORM` element is a block level element - that means that the browser will present it on a new line, like it does with headings and paragraphs.

The `FORM` element's attributes include:

Table 4-1. FORM element attributes

attribute	example	description
`method`	`method="POST"`	the http method to use to send the form's contents back to the web server. it can be `POST` or `GET` -- the differences are explained the the http cheat sheet appendix.
`action`	`action="../cgi-bin/myscript.cgi"`	the relative or absolute URL of the cgi program which is to process the form's data
`enctype`	`enctype="application/x-www-form-urlencoded"`	the encoding type for the document submission. we'll cover this more later.

Printing your form with CGI.pm

To print a form from CGI.pm, we can't write `form(..)` as we'd expect. This is because almost every situation we might deal with wants to start a form, generate form internals and then end the form. Hence, as you might expect we start and end a form as:

```
print start_form({-action=>"myscript.pl", -method=>"POST"});
        # print form internals.
print end_form();
```

CGI.pm FORM defaults

CGI.pm has very reasonable defaults for these form elements. These are as follows:

Table 4-2. CGI.pm FORM defaults

attribute	default
`method`	`method="POST"`
`action`	`action="myscript.cgi"` or whatever your current cgi script is called.
`enctype`	`enctype="application/x-www-form-urlencode"`

We shouldn't ever need to change the `enctype` within our form (in fact we should avoid even trying to do so).

If you want to create a multi-part form (for uploading files etc) we create this in a different manner. We must use this syntax for multi-part forms if we want CGI.pm to handle them properly. We create a multi-part form like this:

```
print start_multipart_form({-action=>"myscript.pl", -method=>"POST"});
        # print form internals.
print end_form();
```

Form input fields and CGI.pm

Some of the input fields you can use in your form include:

TEXT

A text input field `<INPUT TYPE="TEXT" NAME="email_address" VALUE="bob@hotmail.com">`

This can be created by either:

```
print textfield({-name=>"email_address",
                 -value=>"bob\@hotmail.com"});
        # or
print input({-type => "text",
             -name => "email_address",
             -value => "bob\@hotmail.com"});
```

HIDDEN

Hidden fields allow us to pass data around without having to display it to the user. Using hidden doesn't mean that the user *cannot* see the data -- as it's there in the source -- but it means that the user doesn't have to worry about it.

Hidden fields should always have a value defined.

```
<INPUT TYPE="HIDDEN" NAME="stage" VALUE="3">
```

This can be created by either:

```
print hidden({-name=>"stage",
              -value=>"3"});
        # or
print input({-type => "hidden",
             -name => "stage",
             -value=> 3 });
```

PASSWORD

Password fields allow the user to enter a password without fear of on-lookers learning it. Values entered into a password field are obscured with '*'s. It is not common to give a password a value as that value can be seen in plain text if the page source is viewed. It is possible to do it though.

```
<INPUT TYPE="PASSWORD" NAME="passwd" VALUE="enter">
```

This can be created by either:

```
print password_field({-name=>"passwd",
                      -value=>"enter"});
        # or
print input({-type => "password",
             -name => "passwd",
             -value=> "enter"});
```

CHECKBOX

Creates a yes/no checkbox. Saying `CHECKED` will make it checked by default.

```
<INPUT TYPE="CHECKBOX" NAME="send_email" CHECKED>
```

This can be created by either:

```
print checkbox({-name=>"send_email",
                -value=>"ON",
                -checked=>"checked",
                -label=>""});
        # or (so long as we don't want it checked)
print input({-type => "checkbox",
             -name => "send_email"});
```

CHECKBOX GROUP

Creates a group of yes/no checkbox. Note that all we have to do here is print out several checkboxes with the same name. Adding `CHECKED` in any of them will make those checked by default.

```
<input type="checkbox" name="group_name" value="fst" checked />
your first choice<br />
<input type="checkbox" name="group_name" value="snd" />
your second choice<br />
<input type="checkbox" name="group_name" value="thr" />
your third choice<br />
<input type="checkbox" name="group_name" value="fth" checked />
your fourth choice<br />
```

With CGI.pm we can generate these by:

```
my %labels = ('fst'=>'your first choice',
           'snd'=>'your second choice',
           'thr'=>'your third choice',
           'fth' => 'your fourth choice');
print checkbox_group({-name=>'group_name',
                      -values=>['fst','snd','thr','fth'],
                      -default=>['fst','fth'],
                      -linebreak=>'true',
                      -labels=>\%labels});
```

Let's look at this a little bit. Specifying a hash of labels gives us a way of linking the values we want to pass through the form and the values we want the user to see. In this case we want the user to get a list of values such as

- your first choice
- your second choice
- your third choice
- your fourth choice

but we want to pass the values `fst`, `snd`, `thr`, `fth` on if they're selected. We still have to pass in an array of values, because hashes don't have ordering. The array preserves the ordering of the items. If the labels hash is empty or non-existent then the user gets our list of values.

Note that we pass the values and defaults as anonymous array references. If we had arrays with these values in them we could pass references to those in, instead. This is common to many CGI form functions. For example:

```
my %labels = ('fst'=>'your first choice',
           'snd'=>'your second choice',
           'thr'=>'your third choice',
           'fth' => 'your fourth choice');
my @values = ('fst','snd','thr','fth');
my @defaults = ('fst','fth');

print checkbox_group({-name=>'group_name',
                      -values=>\@values,
                      -default=>\@defaults,
                      -linebreak=>'true',
                      -labels=>\%labels});
```

If you're worried about that `linebreak` option, it merely says whether we add `<br />` tags after each checkbox input.

SELECT

Creates a drop-down list of items. Saying `SELECT MULTIPLE` will allow for multiple choices to be made.

```
<SELECT NAME="menu_name">
    <OPTION VALUE="fst">your first choice</OPTION>
    <OPTION VALUE="snd">your second choice</OPTION>
    <OPTION VALUE="thr">your third choice</OPTION>
    <OPTION VALUE="fth">your fourth choice</OPTION>
</SELECT>
```

We have a couple of ways in doing this with CGI.pm. The typical style of SELECT lists are what CGI.pm calls popup menus. To create a popup menu we do:

```
my %labels = ('fst'=>'your first choice',
           'snd'=>'your second choice',
           'thr'=>'your third choice',
           'fth' => 'your fourth choice');
my @values = ('fst','snd','thr','fth');

print popup_menu({-name=>'menu_name',
                  -values=>\@values,
                  -default=>'snd',
                  -labels=>\%labels});
```

The alternative SELECT lists, which allow multiple element selections are called scrolling lists in CGI.pm. To create a scrolling list we do:

```
%labels = ('fst'=>'your first choice',
           'snd'=>'your second choice',
           'thr'=>'your third choice');
@values = ('fst','snd','thr');
@defaults = ('fst','thr');
print scrolling_list({-name=>'list_name',
                      -values=>\@values,
                      -default=>\@defaults;
                      -size=>5,
                      -multiple=>1,
                      -labels=>\%labels});
```

Note that we're allowed to have more than one default in this one because we set `multiple` to true (1). If we set `multiple` to false (0) then each of the elements in `@defaults` will be set as selected, but how this is handled is browser dependent.

SUBMIT

Creates a button which, when pressed, will submit the form.

```
<INPUT TYPE="SUBMIT" VALUE="Press me!">
```

A submit button in CGI.pm is created as follows:

```
print submit({-value=>"Press me!"});
```

If we need to name this button, we can do that too.

 All of these form input types and more can be found by checking out **perldoc CGI**.

Exercises

1. Write a program that uses the CGI module to create:

   - 1 scrolling list with 4 elements
   - 1 popup list with 3 elements
   - A single check box.
   - A text or password field

   Experiment with passing both array references and anonymous array references.

Chapter summary

- To create a HTML form with CGI you use start_form, or start_multipart_form. To close either you use end_form.
- CGI.pm's form defaults return the form submission to your current script by the post method.
- Many form elements in CGI.pm are not named after their HTML counterparts.
- HTML tags which may have multiple inputs (such as selections) are handled in CGI.pm by passing an array reference for values (to define ordering) and a hash reference (to pair up values and labels).

Chapter 5. Accepting and processing form input

In this chapter...

CGI programs are often used to accept and process data from HTML forms. In this section, we show how we can use the `CGI.pm` module to parse form data.

CGI Parameters

One common (but fiddly) task in CGI programming is taking the parameters given in an HTML form and turning them into variables that you can use.

The parameters from an HTML form are encoded in this "url-encoded" format:

```
name=Paul&company=Obsidian%20Consulting%20Group
```

If you use the POST method, these parameters are passed via STDIN to the CGI script, whereas GET passes them via the environment variable `QUERY_STRING`. This means that as well as simply parsing the character string, you need to know where to look for it as well.

The easiest way to parse this parameter line is to use our friendly `CGI.pm` module. If we are passed a parameter `name` by either a GET or POST operation then we can access it using the `param()` function. For example:

```
#!/usr/bin/perl -w

use strict;
use CGI qw/:standard/;

my $name = param('name');

print header;
print start_html;
print "Hello, $name!";
print end_html;
```

This program is in your `cgi-bin` directory as `hello_name.cgi`.

Calling `param()` in context

Certain types of form input fields define multiple values of the same name. For example a checkbox group may have more than one checkbox checked. A scrolling list might have more than one element selected. To access these we ask `param` for an array.

```
        # put all the checkbox values that were checked into @checked.
my @checked = param('group_name');
```

Of course if we know that we'll only ever have one value we can say:

```
my $checked = param('send_email');
```

When we call `param` in a scalar context, we will always get a scalar result. If that parameter was actually given a number of values, we'll just get the first one of them, nothing will tell us that there was more than one. This is a great example of Perl doing what we mean, most of the time.

Where this hurts

Something you're going to want to do, sooner or later, is pass the values from `param()` to a subroutine of yours.

```
check_input(param("name"), param("phone"));
```

This will work every time that `param("name")` and `param("phone")` are defined. However, if ever `param("name")` is undefined, then the first argument passed to `check_input`, will not be the undefined element, but will instead be `param("phone")`. If you attempt to access a second parameter, it will be returned as undef.

This occurs because when we call `param()` within a call to a subroutine we're calling it in list context. If any of our calls to `param()` return the empty list then these disappear when our lists get concatenated and passed to the subroutine.

This also means that if the user managed to send us more than one name, then `param("phone")`'s value would be displaced in our list, and we'd probably ignore it. This is usually much less of a problem, but it might still surprise you occasionally.

The only way to get around this is to make sure that you don't just pass param values into your function. There are lots of ways of avoiding this. Here are two:

```
my $name = param("name");
my $phone = param("phone");
check_input($name, $phone);

        # or
check_input(scalar(param("name")), scalar(param("phone")));
```

I recommend the first.

What were my parameters again?

Let's suppose that we want to get a list of all the parameters we we passed in. Perhaps we're writing a generic program to handle a bunch of different form inputs.

We can get the parameter names as follows:

```
my @parameters = param();
```

That's it. `@parameters` now contains the names of all the parameters that our program was passed. We can now do whatever we want with these, including using a foreach loop to get their values.

Debugging with the `CGI.pm` module's offline mode

As suggested in the previous chapter, when you run a CGI script that makes use of the `CGI.pm` module from the command line, it will ask you for any parameters you wish to pass to the script. The prompt will look like this:

```
(offline mode: enter name=value pairs on standard input)
```

This allows you to enter parameters in the form `name=value` for testing and debugging purposes. Use **CTRL-D** (the Unix end-of-file character) to indicate that you are finished:

```
(offline mode: enter name=value pairs on standard input)
name=fred
age=40
^D
```

CGI.pm assumes that the `value` pairs that you pass it are url-encoded. We're just about to cover how you can url-encode a variable.

If you don't use the `CGI.pm` module, then you're on your own when it comes to testing. Many other methods of processing CGI parameters (including many home-grown ones), *only* work when called from a web server as part of a CGI query. If you want to be able to do automated testing of your scripts without a web server getting involved, it's essential you make use of the `CGI.pm` module.

A nice but sometimes surprising behaviour of the CGI.pm class is to assign parameters from `param` to your input fields. This means that if your script submits to itself and some of the validation fails you can reprint the passed in data with no further effort. On the other hand you may not get the value you expected to come out in your field.

To solve this problem, if you want the value you supply to *always* be the initial value in that input then use the override option:

```
hidden({-name=>"...",
        -value=>"...",
        -override=>1});
```

Exercises

1. Write a simple form to ask the user for their name. Have the action of the form submit the data to `hello_name.cgi` and see if it works.
2. Add some fields to your form, including a checkbox and a popup menu, and print out their values. What are the default true/false values for a checkbox?
3. What happens if you use the a scrolling list form functionality?

   Try assigning that field's parameters from it to an array instead of a scalar, and you will see that the data is handled smoothly by the `CGI.pm` module. Print them out using a `foreach` loop, as in earlier exercises.

Building a GET string

Very occasionally we don't actually want to have the user input data through a form, rather we'd just like to give them a premade link to follow that passes our script any parameters we need. In this case we have to build the GET string ourselves. One thing that we need to make sure of is that the parameters we pass are in a form that our browser will support. So, we have to replace spaces with `%20`s or `+`s and escape other punctuation with the hexadecimal representation of their ASCII values.

Fortunately the `CGI.pm` module is very helpful here, with a function called `escape`. This function has an opposite `unescape` such that:

```
unescape(escape($string)) eq $string
```

is true, but you shouldn't need to use `unescape` all that often.

To build a GET string just do something like the following:

```
my %to_pass = (name =>"Jessie Bottlebrush",
               age=>13,
               fav_colour=>"Orange");

my $get_string = "name=".escape($to_pass{"name"});
$get_string .= "&age=".escape($to_pass{"age"});
$get_string .= "&fav_colour=".escape($to_pass{"fav_colour"});

my $url = a({-href=>"my_script.cgi?$get_string"}, "Click Here");
```

Of course, TMTOWTDI:

```
my $get_string = join("&",map(keys %to_pass,{"$_=".escape($to_pass{$_})}));
```

Try to avoid building up get strings too often though, as it is an ugly way of doing things.

Practical Exercise: Data validation

Your trainer will now demonstrate and discuss the use of `CGI.pm` for validation of data entered into a web form. An example form is in your `public_html` directory as `validate.html` and the validation CGI script is available in your `cgi-bin` directory as `validate.cgi`.

```
#!/usr/bin/perl -w

use strict;
use CGI qw/:standard/;
use CGI::Carp 'fatalsToBrowser';

print header;
print start_html({-title=>"Validation Script"});

my @errors;

push (@errors, "Year must be numeric") if param('year') !~ /^\d+$/;
push (@errors, "You must fill in your name") if param('name') eq "";
```

```
push (@errors, "URL must begin with http://") if param('url') !~ m!^http://!;

if (@errors) {
        print h2("Errors) . "\n";
        print start_ul . "\n";
        foreach (@errors) {
                print li($_) . "\n";
        }
        print end_ul . "\n";
} else {
        print p("Congratulations, no errors!") . \n";
}
print end_html;
```

Exercises

1. Open the form for the validation program in your browser. Try submitting the form with various inputs.

Practical Exercise: Multi-form "Wizard" interface

Your trainer will now demonstrate and discuss how you can use what you've just learnt to create a multi-form "wizard" interface, where values are remembered from one form to the next and passed using hidden fields.

Make sure you use `override=>1` if it's possible that your hidden field name might be accidently overridden by values in the parameter list.

What's more, CGI.pm will get very upset if you declare a hidden field without a value.

Source code for this example is available in your `cgi-bin` directory as `wizard.cgi`.

First, we print some headers and pick up the "step" parameter to see what step of the wizard interface we're up to. We have four subroutines, named `step1` through `step4`, which do the actual work for each step.

```
#!/usr/bin/perl -w

use strict;
use CGI qw/:standard/;
use CGI::Carp 'fatalsToBrowser';

print header, start_html, h1("Wizard interface);

my $step = param('step') || 0;

step1() unless $step;
step2() if $step == 2;
step3() if $step == 3;
step4() if $step == 4;
```

```
print end_html;
```

Here are the subroutines. The first one is fairly straightforward, just printing out a form:

```
#
# Step 1 -- Name
#
sub step1 {
        print h2("Step 1: Name),
                p("What is your name?"),
                start_form,
                     hidden({-name=>"step",
                             -value=>2,
                             -override=>1}),
                     textfield({-name=>"name"}),
                     submit({-value=>"Continue"}),
                end_form;
}
```

Steps 2 through 4 require us to pick up the CGI parameters for each field that's been filled in so far, and print them out again as hidden fields:

```
#
# Step 2 -- Quest
#

sub step2 {
                # Note that we rely on param filling
                # in our hidden name field below.
        print h2("Step 2: Quest"),
              p("What is your quest?"),
              start_form,
                   hidden({-name=>"step",
                           -value=>3,
                           -override=>1}),
                   hidden({-name=>"name"}),
                   textfield({-name=>"quest"}),
                   submit({-value=>"Continue"}),
              end_form;
}


#
# Step 3 -- silly question
#

sub step3 {
                # Note that we rely on param filling
                # in our hidden name and quest fields below.
        print h2("Step 2: Silly Question"),
              p("What is the airspeed velocity of an unladen swallow?"),
              start_form,
                   hidden({-name=>"step",
                           -value=>4,
                           -override=>1}),
                   hidden({-name=>"name"}),
                   hidden({-name=>"quest"}),
                   textfield({-name=>"swallow"}),
                   submit({-value=>"Continue"}),
              end_form;
}
```

Step 4 simply prints out the values that the user entered in the previous steps:

```
#
# Step 4 -- finish up
#

sub step4 {
        my $name = param('name');
        my $quest = param('quest');
        my $swallow = param('swallow');
        print h2("Step 4: Done!"),
              p("Thank you!"),
              p(qq{Your name is $name.  Your quest is $quest.
                The airspeed velocity of an unladen swallow is
                $swallow.});
}
```

Exercises

1. Add another question to the `wizard.cgi` script.

Practical Exercise: File upload

`CGI.pm` can also be used to allow users to upload files. Your trainer will demonstrate and discuss this. Source code for this example is available in your `cgi-bin` directory as `upload.cgi`

First off, you need to specify an encoding type in your form element. The attribute to set is `ENCTYPE="multipart/form-data"`.

To create this kind of form within a CGI script, you must use the functions `start_multipart_form` and `end_form`. This also makes it very clear that you are using a multipart form to your program's future maintainers.

Since this script has no need of being dynamic, we might as well put it all in a HTML file. You'll find it in `upload.html`.

```
<html>
<head>
<title>Upload a file</title>
</head>
<body>
<h1>Upload a file</h1>

Please choose a file to upload:

<form action="upload.cgi" method="POST" enctype="multipart/form-data">
<input type="file" name="filename">
<input type="submit" value="OK">
</form>
</body>
</html>
```

`CGI.pm` handles file uploads quite easily. Just use `upload()` instead of param(). The value returned is special -- in a scalar context, it gives you the filename of the file uploaded, but you can also use it in a filehandle.

```
#!/usr/bin/perl -w

use strict;
use CGI qw/:standard/;

my $filename = upload('filename');
my $outfile = "outputfile";

print header;

# There will probably be permission problems with this open
# statement unless your script is setuid or $outfile is world writable.
# There are potential security issues as well.
# But let's not worry about all that for now.

open (OUTFILE, ">$outfile") || die "Can't open output file: $!";

{
        local $/="";    # Prepare to read in the whole file at once.
                        # Here we use $filename as a filehandle
        my $filecontents = <$filename>; # read in the whole file

        print OUTFILE $filecontents; # Print everything to OUTFILE
}

close OUTFILE || die "Can't close OUTFILE: $!";

print p("Uploaded file and saved as $outfile");

print end_html;
```

This code can be found in `upload.cgi`.

Chapter summary

- The `CGI.pm` module can be used to access parameters passed to the CGI program using the `param()` function.
- Using the `param()` function in a list context will return all of the values passed to the program with that key.
- Care should be taken if `param()` is ever going to be passed to subroutines.
- Calling `param()` in a list context without a key will return all of the names of the name=value pairs.
- CGI.pm will fill in all of your form input fields with values from `param()` if possible. To prevent this you have to use the `override=>1` option in your input field.
- File uploads must use multipart forms.
- To access the file from an upload call the `upload()` function rather than the `param()` function. The return value can be used as both the filename and a filehandle.

Chapter 6. Security issues

In this chapter...

In this section we briefly examine some security issues arising from the use of CGI scripts, including authentication and access control, and the risk of tainted data and how to avoid it.

This is not a complete guide to CGI security, but rather a simple discussion of a few important points. Just because you follow all the recommendations in this chapter does not mean that your script is free of security flaws.

Authentication and access control for CGI scripts

A common question asked by new CGI programmers is "How do I protect my web site with a CGI script?" There are various ways to use CGI programs to ask for usernames and passwords and perform authentication, but in fact the best way to perform authentication and access control comes with your web server and doesn't require any programming at all.

The reason that password protection is often connected with CGI programs is that CGI programs are more likely to interact with the web server's underlying file system, backend databases, or other things which need to be kept secure. Many programmers assume that because CGI can be used for password protection, it is the right choice for the job. This is not always true.

One of the best ways to password protect web pages is by using the web server's own authentication and access control mechanisms. Since we're using the Apache web server, we'll look at how to do it with that.

Why is CGI authentication a bad idea?

Authentication (i.e. username and password checking) is hard to do correctly in CGI. Some common pitfalls include:

- Username and password strings are sent as parameters in a GET query, and end up in the URL (eg `http://example.com/my.cgi?username=fred&password=secret`). These details can then end up in peoples' bookmark files, other sites' referrer logs, and so on.
- Sometimes username and password details are passed back and forth using "cookies". Some users choose to have cookies disabled due to privacy concerns, and the website will therefore be unusable to them. No such problem exists with HTTP authentication via the web server

HTTP authentication

If a web page or CGI script requires a username and password to view it, the HTTP conversation between the client and the server goes like this:

1. The user specifies a URL
2. The user agent connects to port 80 of the HTTP server
3. The user agent sends a request such as `GET /index.html`

4. The user agent may also send other headers
5. The HTTP server realises that authentication must be performed {usually by looking up configuration files}
6. The HTTP server returns a status code 401, meaning "Unauthorised", and also a header saying `WWW-Authenticate:` and the name of the authentication domain, for instance "Acme Widget Co. Staff". This usually appears in the browser's dialog box as "Please provide a username and password for Acme Widget Co. Staff".
7. The browser presents a dialog box or other means by which the user can enter their username and password, which the user fills in then clicks "OK"
8. The browser sends a new request, this time including an extra header saying `Authorisation:` and the appropriate credentials
9. If the HTTP server finds that the credentials are valid, it sends back the resource requested and closes the connection
10. Otherwise, it sends back another response with status code `401` (and probably a body containing an error message), which the user agent should recognise as meaning that the authentication failed, and display the body.

Why is HTTP authentication a bad idea?

HTTP authentication (also known as *HTTP Basic Auth*) has its own disadvantages. These include:

- The login interface cannot be configured. Many people feel this detracts from the look and feel of a website.
- Authentication tokens remain active until the user's browser shuts down, which can be an issue in public computer labs and other situations where computers may be shared.
- Unlike cookies, which can be stored over multiple sessions, the user needs to enter their username and password each session when using HTTP authentication. Some newer browsers, however, will remember these details for the user.
- The user needs to actually have a username and password in order for HTTP authentication to work. This means we cannot provide customisation for an anonymous user visiting our site.

Access control

The way access control (using HTTP authentication) is handled varies from one web server to another. If your web server is not Apache, you will need to contact your web server administrator or read the documentation it came with, as only Apache is covered in this course.

Apache implements HTTP authentication with the use of a password file and either server configurations or a special file (usually called `.htaccess`) in the web directory, which can override the server configuration.

Whether or not you can use `.htaccess` files, and what options they honour, varies from server to server. Since they are not part of Perl, we will not cover them further in this course.

Tainted data

Sometimes you will want to write a CGI script which interacts with the system. This can result in major security risks if the commands executed on the system are based on user input. Consider the example of a finger program which asked the user who they wanted to finger.

```
#!/usr/bin/perl -w

use strict;

print "Who do you want to finger? ";
my $username = <STDIN>;
print ‘finger $username‘;       # backticks used to execute shell command
```

Imagine if the user's input had been `pjf; cat /etc/passwd`, or worse yet, `pjf; rm -rf /` The system would perform both commands as though they had been entered into the shell one after the other.

Luckily, Perl's `-T` flag can be used to check for unsafe user inputs.

```
#!/usr/bin/perl -wT
```


Documentation for this can be found by running **perldoc perlsec** section of the online documentation, or on page 557 of the Camel book (page 356 2nd Ed.).

`-T` stands for "taint checking". Data input by the user, either via the command line or an HTML form, is considered "tainted", and until it has been modified by the script, may not be used to perform shell commands or system interactions of any kind.

The only thing that will clear tainting is referencing substrings from a regexp match. **perldoc perlsec** contains a simple example of how to do this, about 7 pages down. Read it now, and use it to complete the following exercises.

Note that you'll also have to explicitly set `$ENV{'PATH'}` to something safe (like `/bin`) as well.

Exercises

1. The HTML file `finger.html` asks the user for an account name about which to obtain information {using the Unix system's `finger` command}. It calls the CGI script `cgi-bin/finger.cgi` which uses taint checking.
2. Why is the data input by the user tainted?
3. Add a `-T` flag to the shebang line of `finger.cgi` so that the script performs taint checking
4. Try re-submitting the form - it should fail
5. To untaint the data, you need to clean up any unwanted characters. Use some code similar to that in **perldoc perlsec** to remove anything other than alphanumeric characters and assign the valid part of the user input to a new variable.

Secure HTTP

Another somewhat related topic is secure HTTP, which uses the HTTPS protocol to open a secure connection and encrypts all data between the web client and server. This is often used to make transactions involving private information (such as credit card details) more secure.

CGI scripts can be run on a secure server exactly as they would run on any other server.

Chapter summary

- HTTP authentication can be used to password protect web pages
- The Apache web server implements HTTP authentication. This can be configured via a `.htaccess` file
- There is a security risk from tainted data --- data entered by a user which is used for subsequent system interaction
- Perl has built-in checking for tainted data, which can be turned on my using the `-T` command line switch
- Data can be untainted by referencing a substring in a match, as shown in **perldoc perlsec**.
- Secure HTTP can be used to provide an encrypted channel of communication between the web client and server.

Chapter 7. Splitting HTML and code with HTML::Template

In this chapter...

Embedding HTML inside your Perl script can make it difficult to maintain both your HTML and your code. This is especially the case if somebody else is writing the HTML and may wish to change it at a later date.

To avoid the issue of mixing code and HTML, Perl has a number of useful templating modules which can be used to keep things separate. These have a great many advantages -- it's easy to change the interface, or have multiple interfaces available. Web-designers and programmers are less likely to step on each others toes, and people can use the most appropriate tools for each part. This chapter will explain the use of the `HTML::Template` module, although you should be aware that other templating modules are available.


The HTML::Template module does not come standard with Perl, but can be easily downloaded from CPAN (the Comprehensive Perl Archive Network) (http://www.cpan.org/). You can get documentation for it by reading **perldoc HTML::Template**.

What is HTML::Template

Like the `CGI` module, `HTML::Template` is a module to help make your life easier when writing CGI scripts. Instead of embedding HTML into your code, `HTML::Template` allows you to load a custom template or blueprint and fill in special fields. If used properly, `HTML::Template` can eliminate the need to have any HTML in your script at all. [1]

Things are best illustrated with an example. Here's a simple template that prints a simple library-book reminder.

```
<HTML>
<HEAD><TITLE>Library reminder</TITLE></HEAD>
<BODY>
Dear <TMPL_VAR name="name">,
<P>
Don't forget that your book titled <TMPL_VAR name="title">
by <TMPL_VAR name="author"> is due back
<TMPL_VAR name="duedate">.
<P>
If your book is returned late, a fine of $<TMPL_VAR name="fine">
will apply for each <TMPL_VAR name="timeperiod"> the book
is late.
<P>
Yours sincerely,
<BR>
   <I>The management</I>.
</BODY>
</HTML>
```

All those funny TMPL_VAR tags are used by HTML::Template, and get replaced with text supplied by the program at execution time. Here's a script that uses the template we've just seen to print a library reminder.

```
#!/usr/bin/perl -w
use strict;
use HTML::Template;

my $template = HTML::Template->new(filename => "library.html");

$template->param(
       name       => "Paul Fenwick",
       title      => "Programming Perl, 3rd Ed",
       author     => "Larry Wall, Tom Christiansen and Jon Orwant",
       date       => "next Wednesday",
       fine       => 2.20,
       timeperiod => "week");

print "Content-Type: text/html\n\n",$template->output;
```

Yes, it really is that simple. Since HTML::Template let's us split the HTML from the programming interface, we'll talk about them separately.

The Template Explained

HTML::Template provides a very powerful templating mechanism with many features more than just simple variable substitution. In this section we'll talk about these features, starting from the simple ones and proceeding onto more advanced topics.

Conventions

We've already seen one special tag that HTML::Template allows us to use, and that's <TMPL_VAR>. With our example above, we used it just like a normal HTML tag. That's great if we're using an HTML editor that's happy with those funny looking tags, but what if we're using programs that are not? We might want to run our templates through a validation service, in which case any template-tags are likely to cause problems.

To get around this, HTML::Template also allows us to embed HTML-style comments which operate the same way as the standard template-tags. We use either comment-style or tag-style templating methods, and we can mix both styles in the same document if we desire. Here's the example above using comment-style templating.

```
<HTML>
<HEAD><TITLE>Library reminder</TITLE></HEAD>
<BODY>
Dear <!-- TMPL_VAR name="name" -->,
<P>
Don't forget that your book titled <!-- TMPL_VAR name="title" -->
by <!-- TMPL_VAR name="author" --> is due back
<!-- TMPL_VAR name="duedate" -->.
<P>
If your book is returned late, a fine of $<!-- TMPL_VAR name="fine" -->
will apply for each <!-- TMPL_VAR name="timeperiod" --> the book
is late.
```

```
<P>
Yours sincerely,
<BR>
   <I>The management</I>.
</BODY>
</HTML>
```

For these notes, we'll always use template-tags as we did so with the example at the start of the chapter. We adopt this approach because it's more compact, and (depending upon how you are reading this document) should help save on bandwidth and trees.

There's another note of convention that we need to mention before we progress any further, and that's of filenames. `HTML::Template` doesn't care what the name of a template file is [2] so we can use anything we like. Programmers traditionally prefer to use files ending in `.tmpl` as it makes it obvious that they're templates. Web-designers, on the other hand, tend to prefer `.html` because it means their favourite HTML-editor is more likely to play nicely with the file. In these notes, we'll use `.html` -- after all, a plain HTML file is just a `HTML::Template` file without any special tags.

Simple Template Fields

With the code already presented we've shown how to use `TMPL_VAR` fields as place holders for data that we'll plug into the document at run-time. How they work should be fairly self-explanatory -- the tag is removed and the data we supply is inserted into its place.

`TMPL_VAR` and other templating tags don't need to obey the regular rules of HTML. For example, it's perfectly valid to have a template tag inside an HTML tag. The following code lets us set the alt tag on an image at run-time.

```
<IMG SRC="/images/picture.jpg" ALT="<TMPL_VAR NAME=foo>">
```

It's usually considered good style to quote values inside HTML and templating tags. In our example immediately above, we intentionally wrote `NAME=foo` rather than `NAME="foo"`. The second case runs the risk of seriously confusing any HTML editors we choose to use, whereas the first just it appear that we just have a very odd `ALT` value.

Exercises

1. Take the HTML in `public_html/petpage.html` and modify it to insert templating fields for `name`, `age` and `pet`.
2. Check out `cgi-bin/petpage.cgi` to see the effects of your changes (view it in your browser).
3. Now take that file and modify it so that the background colour can be set dynamically. Use the templating field `background`.
4. Check out `cgi-bin/petpage.cgi` to see the effects of your changes (view it in your browser).

Escaping in template fields

Sometimes we want to don't want our data to appear verbatim inside the HTML that we're producing. This is particularly the case if we're inserting data that might contain less-than or greater-than signs, or other characters that have special meaning. In this case we want to do *HTML encoding* Sometimes we want to encode information into a URL, in which case we want to do *URL encoding*. Sometimes we might even want to display data in its encoded and unencoded forms in the same page.

Rather than having to do this tedious escaping ourselves in our Perl code, we can get `HTML::Template` to do the hard work for us. This is also best illustrated by example.

```
This is how I escape for HTML  <TMPL_VAR name="data" escape="html"><BR>
This is how I escape for a URL <TMPL_VAR name="data" escape="url"><BR>
Here is my data with no escaping  <TMPL_VAR name="data"><BR>
```

If you're using the `CGI` module, then usually you don't have to worry about using these escapes. When used as described previously in these notes, the `CGI` module will usually use the appropriate type of escaping needed for the task at hand.

Conditionals

Sometimes you'll want to display different pages depending upon the execution of your program. In some cases we might select a template to use at runtime, depending upon if, for example, our user was borrowing or returning a book. In other cases, we might want to display fundamentally the same page, but choose to add or remove some sections depending upon circumstances. With our library example, we might want to display a reminder to the user if they have a book that's overdue, or alert them that a book they've placed on hold is available for borrowing.

We could use a `<TMPL_VAR>` tag which we can then bind to either the empty string or the HTML which contains our reminder message and associated formatting. That will work, but it potentially means having ugly chunks of HTML in our code, especially if the reminder comes wrapped in a table with images and special fonts. We started using `HTML::Template` to avoid this very situation, so isn't there a better way?

The solution is to use `HTML::Template`'s conditional tags. Here's our example above with an optional section that only gets displayed if a reminder message exists.

```
<HTML>
<HEAD><TITLE>Library reminder</TITLE></HEAD>
<BODY>
<TMPL_IF name="reminder">
        <CENTER><B>REMINDER</B></CENTER>
        <TMPL_VAR name="reminder">
        <HR>
        <BR>
</TMPL_IF>
Dear <TMPL_VAR name="name">,
<P>
Don't forget that your book titled <TMPL_VAR name="title">
by <TMPL_VAR name="author"> is due back
<TMPL_VAR name="duedate">.
<P>
If your book is returned late, a fine of $<TMPL_VAR name="fine">
will apply for each <TMPL_VAR name="timeperiod"> the book
is late.
<P>
Yours sincerely,
```

```
<BR>
   <I>The management</I>.
</BODY>
</HTML>
```

`HTML::Template` uses the same rules for truth as does regular Perl. As you might expect, there's also `<TMPL_ELSE>` and `<TMPL_UNLESS>` tags too. Unfortunately there is no such thing as a `<TMPL_ELSIF>` tag.

There is no such thing as a `</TMPL_ELSE>` (close TMPL_ELSE) tag, instead you should close with the same tag that you used to open the conditional. That means that every `<TMPL_IF>` needs to have a matching `</TMPL_IF>`, and every `<TMPL_UNLESS>` needs a matching `</TMPL_UNLESS>`, regardless of whether you use `<TMPL_ELSE>` tags or not.

Exercises

1. Take the template in `public_html/tmpl-cond.html` and modify it so that it displays some extra text and the contents of the `error` parameter only if it exists.
2. Use the `public_html/cgi-bin/tmpl-cond.cgi` script to test your changes.

Looping constructs

The examples that we've seen so far are great if we're dealing with singular pieces of data, but what if we want to display a list of books a user has currently borrowed? This list could contain any number of books. How would we write a template to deal with that?

A naive approach would be to use a `<TMPL_VAR>` tag where we wish to insert the list, then build that list and bind it into place. That's great, except now our program has the work of doing the HTML mark up for the list, and that's something we're trying to avoid. To solve this, we need to be able to deal with loops in our templates.

Loops in `HTML::Template` are almost identical in concept to Perl's `foreach` loops -- that is, we step through a list of values, examining the next one in our list every time we go through the loop. Let's see an example.

```
<UL>
        <TMPL_LOOP name="books_borrowed">
        <LI><I><TMPL_VAR name="author"></I>
                  <B><TMPL_VAR name="title"></B>
                  (<TMPL_VAR name="publisher">)
        </TMPL_LOOP>
</UL>
```

Here we've printed an unordered list of books, with some special formatting for author, title and publisher. If we wanted to, we could have the title of the book come first, or we could print the books out in a table, or do any other formatting change, all without having to touch our Perl code at all.

These looping constructs can be very powerful. Your template can be set up to perform different actions for the first and last lines of your loop (for example, opening and closing table tags), and can distinguish between odd and even rows (for example, in case you want alternating rows to have different backgrounds). It's even possible to have conditional constructs based upon whether or not a given loop is empty or not. While the coverage of these concepts is beyond the scope of this course, all the information can be found using **perldoc HTML::Template**.

Including files

Often you'll be working on a website that has elements that are common to every page, like headers or footers, or huge animated advertisements with musical scores. If people are sensible, these common elements are usually placed into separate files and then pasted into the HTML using some mechanism depending upon your web-server or operating environment.

Now, it wouldn't it be nice if we could include these files when using our templates as well? Well, there's a better way than loading the contents into a `<TMPL_VAR>` tag, and that's using a `<TMPL_INCLUDE>` tag. Let's see one in operation.

```
<TMPL_INCLUDE name="header.html">
Thank-you for flying with <TMPL_INCLUDE name="airline">
<TMPL_INCLUDE name="footer.html">
```

`<TMPL_INCLUDE>` includes the file contents as if it were cut'n'pasted directly into the parent file at that point. This means that your include files can include templating information (including further include directives), just like your parent file. Since included files can include other files, there's potential to get into trouble with files endlessly including each other. `<HTML::Template>` provides some protection to this by only allowing includes 10 levels deep, although you can change or disable that if you like.

Using Template Objects

Now, you've all had some experience with writing templates, and as you can see it's possible to do this without any understanding of what the code that processes these templates looks like. That's an important thing to remember, someone doesn't need to know Perl (or any programming language) to create or edit a template. That's why they're good.

In this section, we'll cover what the programmer needs to know in order to have templates work the way they expect. If there's time, we'll even discuss some more advanced templating features that can improve performance or assist in error handling.

Binding simple parameters

We've already seen a script that binds values to parameters in a template. We use the `param` method to set values. While this has the same name and a similar function to that of the `CGI` module, don't be fooled -- the way it processes arguments is subtly different.

To bind a value to a parameter, we pass in that parameter's name and its value, like this:

```
#!/usr/bin/perl -w
use strict;
use HTML::Template;

my $template = HTML::Template->new(filename => "library.html");

$template->param(title => "Programming Perl, 3rd Ed");
$template->param(author => "Larry Wall, Tom Christiansen and Jon Orwant");
```

As you can see from the example above, there's no need to set all the parameters in the same call, you can figure out parameters and set them as you go. If you do want to set a number of values at once, you can, just pass in as many name-value pairs as you need:

```
$template->param(
       name       => "Paul Fenwick",
       title      => "Programming Perl, 3rd Ed",
       author     => "Larry Wall, Tom Christiansen and Jon Orwant",
       date       => "next Wednesday",
       fine       => 2.20,
       timeperiod => "week");
```

You might have realised that these name-value pairs look awfully familiar to things we put into (and take out of) hashes. If you already have a hash of all the data you need, you can plug that directly into the `param()` method and things will work how you'd expect:

```
$template->param(%info);
```

If you try to bind a parameter that doesn't exist in the template you're using, an exception will be thrown (usually resulting in your script dying with an appropriate error). Often this is what you want, as it makes typos immediately obvious.

Sometimes you specifically *don't* want this behaviour. For example, you might write a subroutine which fills in information about the current user. The subroutine would like to provide that information without caring that the template will use all of it, and having your script die just because you didn't want to show the user's age can be a major headache. In these cases, you can request that `HTML::Template` just ignore parameters that don't exist. This is requested at the time you create the template, like this:

```
my $template = HTML::Template->new(filename=>"invite.html",
                                   die_on_bad_params => 0);
```

Binding complex parameters

We've seen how to deal with simple parameters, which are great for dealing with singular pieces of data, but they don't answer what we need to do for loops. That needs something a little bit more

complex. We still use `param` to bind values, but instead of binding each parameter to a single value, we instead bind it to a list reference.

Now, we can't just use any old list reference. You see, `HTML::Template` lets us set a whole swag of different variables each time we go around one of its loops, and a simple list like `[3,4,6,7]` only contains a single value in each position. What we instead want to use is a list of hash references, because a hash *can* contain multiple name-value pairs.

Relax, it sounds difficult, but it's really quite simple, especially when you have an example to work by.

```
$template->param(library_books =>
        [
                {
                        title => "Programming Perl",
                        author => "Larry Wall, et al"
                },
                {
                        title => "Object Oriented Perl",
                        author => "Damian Conway"
                }
        ]);
```

In the example above, our loop would have two iterations. The first time around we'd be dealing with the "Programming Perl" book, and the second time the "Object Oriented Perl" book. If you feel comfortable with references, you can build up this structure in other ways.

Exercises

This should bring all of these concepts together.

1. In `public_html/total.html` you'll find a HTML template. This prints out the headers and footers, and an empty table. Add a looping construct to the table body so that we can fill in the table.
2. In `cgi-bin/total.cgi` you'll find scaffolding for this exercise. This scaffolding includes a number of hash references. Use these to populate the table from the above exercise.
3. You'll notice some more template variables in `public_html/total.html`, pick some values and set these from your cgi program.

Associating other objects

Wow, this `HTML::Template` module is great stuff. I've got a CGI script which takes input from a user, and then displays some or all of it back on a confirmation page along with some other details. Being a good programmer, I'm much too lazy to pull everything out from my `CGI` object and push them back into my `HTML::Template` object. Is there any way I can do this automatically?

It so happens that `HTML::Template` allows you to *associate* a template with another object which has a parameter list, such as a `CGI` object. This means that if you don't provide a value for a given parameter, the value on the *associated object* will be used instead.

So, say that a user has filled in their name, address, phone number, and number of plush penguin toys they own. Rather than having code like this:

```
#!/usr/bin/perl -w
use strict;
use CGI;
use HTML::Template;

my $cgi = CGI->new;
my $template = HTML::Template->new(filename => "penguinrego.html");

$template->param("name",$cgi->param("name"),
                 "address",$cgi->param("address"),
                 "phone",$cgi->param("phone"),
                 "penguins",$cgi->param("penguins"));
```

We can instead have code that looks like this:

```
#!/usr/bin/perl -w
use strict;
use CGI;
use HTML::Template;

my $cgi = CGI->new;
my $template = HTML::Template->new(filename => "penguinrego.html",
                                   associate => $cgi);
```

That's it. Fields we don't fill in explicitly just get copied out of our CGI object without any extra work on our behalf. Fields that we do fill in ourselves will have those values. Nifty, isn't it?

Chapter Summary

- `HTML::Template` allows you to split your Perl code from your HTML code.
- You can use either standard tags or *comment tags* for writing templating fields. You can mix both in the same document.
- Template fields can be used to escape the text which is bound to them for both inclusion in HTML and in URLs.
- `HTML::Template` supports conditionals and loops, which are particularly useful for generating tables.
- `HTML::Template` can be used to include files into your HTML. These files are also evaluated for templating tags.
- To bind values to loops, we need to pass `HTML::Template` a list reference containing many hash references.
- It's possible to *associate* a `CGI` object with a `HTML::Template` object, to have parameters submitted by the user automatically filled in.

Notes

1. Actually, you can remove that unsightly Perl code from your script as well. Buy your trainer a couple of drinks after work and they might show you how. If your trainer is too weird and you don't want to buy them a drink, try looking at the Acme::Bleach (http://search.cpan.org/search?mode=module&query=Acme%3A%3ABleach) module if it's installed on your system.
2. In fact, you don't even have to use filenames at all. You can pass `HTML::Template` other things to use as templates. See **perldoc HTML::Template** for more details.

Chapter 8. Conclusion

What you've learnt

Now you've completed Perl Training Australia CGI Programming in Perl module, you should be confident in your knowledge of the following fields:

- What CGI is
- How the Hypertext Transfer Protocol (HTTP) allows web user agents (browsers) to communicate with web servers and retrieve documents
- How to generate simple web pages using Perl's `CGI` module
- How to access environment variables from CGI scripts
- How to use the `qq()` type functions to quote text
- How to process data from HTML forms using the CGI module
- How to use the CGI module for applications such as data validation, simple "wizard" interfaces, and file uploads
- Security issues related to CGI programming, including authentication and access control, dealing with tainted data, secure web servers, etc.
- How to use `HTML::Template` to separate your HTML and your code.

Where to now?

To further extend your knowledge of Perl, you may like to:

- Work through the material included in the appendices of this book.
- Visit the websites in our "Further Reading" section (below)
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme"
- Install Perl and a web server such as Apache on your home or work computer
- Practice using Perl for CGI programming on a daily basis
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)
- Join an on-line Perl community such as PerlMonks (http://www.perlmonks.org/)
- Extend your knowledge with further Perl Training Australia courses such as:
  - Database Programming with Perl
  - Perl Security
  - Object Oriented Perl

  Information about these courses can be found on Perl Training Australia's website (http://www.perltraining.com.au/).

Further reading

Books

- The `CGI` module's documentation (http://stein.cshl.org/WWW/software/CGI/)
- Scott Guelich et al, *CGI Programming with Perl* (2nd Ed), O'Reilly and Associates, 2000. ISBN 1-56592-419-3.
- Tom Christiansen and Nathan Torkington, *The Perl Cookbook*, O'Reilly and Associates, 1998. ISBN 1-56592-243-3.
- Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 1997. ISBN 1-56592-257-3.
- Joseph N. Hall and Randal L. Schwartz *Effective Perl Programming*, Addison-Wesley, 1997. ISBN 0-20141-975-0.

Online

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlMonks online community (http://www.perlmonks.org/)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup
- Comprehensive Perl Archive Network (http://www.cpan.org)

Appendix A. Unix cheat sheet

A brief run-down for those whose Unix skills are rusty:

Table A-1. Simple Unix commands

Action	Command
Change to home directory	**cd**
Change to *directory*	**cd *directory***
Change to directory above current directory	**cd ..**
Show current directory	**pwd**
Directory listing	**ls**
Wide directory listing, showing hidden files	**ls -al**
Showing file permissions	**ls -al**
Making a file executable	**chmod +x *filename***
Printing a long file a screenful at a time	**more *filename*** or **less *filename***
Getting help for *command*	**man *command***

Appendix B. Editor cheat sheet

This summary is laid out as follows:

Table B-1. Layout of editor cheat sheets

Running	Recommended command line for starting it.
Using	Really basic howto. This is not even an attempt at a detailed howto.
Exiting	How to quit.
Gotchas	Oddities to watch for.
Help	How to get help.

vi (or vim)

Running

```
% vi filename

      or

% vim filename (where available)
```

Using

- `i` to enter insert mode, then type text, press **ESC** to leave insert mode.
- `x` to delete character below cursor.
- `dd` to delete the current line
- Cursor keys should move the cursor while *not* in insert mode.
- If not, try `hjkl`, `h` = left, `l` = right, `j` = down, `k` = up.
- `/`, then a string, then **ENTER** to search for text.
- `:w` then **ENTER** to save.

Exiting

- Press **ESC** if necessary to leave insert mode.
- `:q` then **ENTER** to exit.
- `:q!` **ENTER** to exit without saving.
- `:wq` to exit with save.

Gotchas

vi has an insert mode and a command mode. Text entry only works in insert mode, and cursor motion only works in command mode. If you get confused about what mode you are in, pressing **ESC** twice is guaranteed to get you back to command mode (from where you press i to insert text, etc).

Help

`:help` **ENTER** might work. If not, then see the manpage.

nano (pico clone)

Running

```
% nano -w filename
```

Using

- Cursor keys should work to move the cursor.
- Type to insert text under the cursor.
- The menu bar has ^X commands listed. This means hold down **CTRL** and press the letter involved, eg **CTRL-W** to search for text.
- **CTRL-O** to save.

Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

Gotchas

Line wraps are automatically inserted unless the -w flag is given on the command line. This often causes problems when strings are wrapped in the middle of code and similar.

Help

CTRL-G from the main menu, or just read the menu bar.

Appendix C. ASCII Pronunciation Guide

Table C-1. ASCII Pronunciation Guide

Character	Pronunciation
#	hash, pound, sharp, number
!	bang, exclamation
*	star, asterisk
$	dollar
@	at
%	percent, percentage sign
&	ampersand
"	double-quote
'	single-quote, tick, forward tick
( )	open/close parentheses, round brackets, bananas
<	less than
>	greater than
-	dash, hyphen
.	dot
,	comma
/	slash, forward-slash
\	backslash, slosh
:	colon
;	semi-colon
=	equals
?	question-mark
^	caret (pron. carrot), hat
_	underscore
[ ]	open/close bracket, square bracket
{ }	open/close curly brackets, brace
|	pipe, vertical bar, bar
~	tilde, wiggle, squiggle
`	backtick

Appendix D. HTML Cheat Sheet

The following table outlines a few HTML elements which may be useful to you. For more detail or for information about elements which are not listed here, consult one of the references listed below.

Table D-1. Basic HTML elements

Type of information	Markup
Paragraph	`<P> ... </P>`
Heading level 1	`<H1>This is a level 1 heading</H1>`
Heading level 2	`<H2>This is a level 2 heading</H2>`
Heading level 3	`<H3>This is a level 3 heading</H3>`
Heading level 4	`<H4>This is a level 4 heading</H4>`

Unordered (bulleted) list	`<UL>`
	`<LI>List item 1`
	`<LI>List item 2`
	`<LI>List item 3`
	`<LI>List item 4`
	`</UL>`

Ordered (numbered) list	`<OL>`
	`<LI>List item 1`
	`<LI>List item 2`
	`<LI>List item 3`
	`<LI>List item 4`
	`</OL>`

Table	`<TABLE BORDER>`
	`<TR>  <-- "table row" -- >`
	`<TH>Heading for column 1</TH>`
	`<TH>Heading for column 2</TH>`
	`<TH>Heading for column 3</TH>`
	`</TR>`
	`<TR>  <-- "table row" -- >`
	`<TD>Data for row 1, column 1</TD>`
	`<TD>Data for row 1, column 2</TD>`
	`<TD>Data for row 1, column 3</TD>`
	`</TR>`
	`<TR>  <-- "table row" -- >`
	`<TD>Data for row 2, column 1</TD>`
	`<TD>Data for row 2, column 2</TD>`
	`<TD>Data for row 2, column 3</TD>`
	`</TR>`
	`</TABLE>`

Horizontal rule	`<HR>`
Anchor tag (hypertext link)	`<A HREF="http://example.com/">Descriptive text</A>`

For more information...

- HTMLhelp.org (http://htmlhelp.org/)
- The World Wide Web Consortium (W3C) (http://w3.org/)

Appendix E. HTTP Terminology and reference

Terminology

authentication

: The process by which a client sends username and password information to the server, in an attempt to become authorized to view a restricted resource.

client

: An application program that establishes connections for the purpose of sending requests.

Content-type

: The media type of the body of the response, as given in the `Content-type:` header. Examples include `text/html`, `text/plain`, `image/gif`, etc.

method

: Indicates what the server should do with a resource. Case sensitive. Valid methods include: GET, HEAD, POST

request

: An HTTP request message sent by a client to a server

resource

: A network data object or service which can be identified by a URI.

response

: An HTTP response message sent by a server to a client

server

: An application program that accepts connections in order to service requests by sending back responses.

status code

: A 3-digit integer indicating the result of the server's attempt to understand and satisfy the request. A table of status codes and their meanings appears below.

Uniform Resource Identifier (URI)

: URIs are formatted strings which identify - via name, location, or any other characteristic - a network resource.

Uniform Resource Locator (URL)

: A web address. May be expressed absolutely (eg `http://www.example.com/services/index.html`) or in relation to a base URI (eg `../index.html`) See also URI.

user agent

The client which initiates a request. These are often browsers, editors, spiders (web-traversing robots) or other end-user tools.

HTTP status codes

Table E-1. HTTP status codes

Code	Meaning
200	OK
201	Created
202	Accepted
204	No Content
301	Moved Permanently
302	Moved Temporarily
304	Not Modified
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable

Colophon

```
Just An Undead Perl Hacker
```

The script on the front cover was written by Joe Lesko for Halloween 2001 as a variant of the traditional 'Just Another Perl Hacker' obfuscated script. It was originally posted on PerlMonks (http://perlmonks.org/), an on-line community where one can find Perl-related discussion, advice, code samples, and even poetry. Joe Lesko's home page can be found at http://joelesko.com/

The code in its native environment can be found at:
http://www.perlmonks.org/index.pl?node_id=117294