CGI Programming in Perl

Kirrily Robert

CGI Programming in Perl

by Kirrily Robert

Open Publications License 1.0

Table of Contents

List of Tables

Chapter 1. Introduction

Welcome to Netizen’s *CGI Programming in 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.

1.1. Course outline

- What is CGI? (60 minutes)
- Generating web pages with a Perl script (45 minutes)
- Practical exercises (45 minutes)
- Accepting and processing form input with the CGI module (60 minutes)
- *Lunch break*
- Practical examples (60 minutes)
- Security issues (45 minutes)
- Other related features and Perl modules (60+ minutes)

1.2. 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, ordered and 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 Netizen’s "Introduction to Perl" course, which many of you will have attended recently. Lastly, an HTML cheat-sheet is provided in Appendix D for those who need reminding.

1.3. 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

1.4. 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 Netizen’s Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs signficantly 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.

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.

1.5. 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 ap-
pears 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.

1.6. Other materials

In addition to these notes, you should have a copy of the required text book for this course: Programming Perl (2nd ed.) by Schwartz, Wall, and Christiansen (published by O'Reilly and Associates) -- more commonly referred to as "the Camel book". The Camel book will be used throughout the day, and will be a valuable reference to take home and keep next to your computer.

Lastly, you will have been given a nametag with your name and company on the front, and a username and password on the back.

Chapter 2. What is CGI?

2.1. 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.

2.2. 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.

2.3. 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 Monash University (ftp://ftp.monash.edu.au/pub/rfc/).

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 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 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

6. The output includes HTTP headers
7. The HTTP server sends back the output of the program
8. The HTTP server closes the connection

2.3.1. 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.

2.3.2. HTTP status codes

Table 2-1. HTTP status codes

Code	Meaning
200	OK
201	Created
202	Accepted
204	No Content

Code	Meaning
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

2.3.3. HTTP Methods

2.3.3.1. 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.

2.3.3.2. 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.

2.3.3.3. 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

2.3.4. Exercises

The HTTP request/response process is usually transparent to the user. To see what's going on, let's connect directly to the web server and see what happens.

Login to the system as for the Introduction to Perl course:

1. Open the telnet program, TeraTerm
2. Connect to the training server (your instructor will give you the hostname or IP number)
3. Login using the username and password you were given
4. From the Unix command line, type **telnet localhost 80** -- this connects to port 80 of the server, where the HTTP daemon (aka the web server) is listening. You should see something like this:

   ```
   training:~> telnet localhost 80
   Trying 1.2.3.4
   Connected to training.netizen.com.au.
   Escape character is '^]'.
   ```

5. Ask the web server for a static document by typing: `GET /index.html HTTP/1.0` then press enter twice to send the request. Note that this command is *case sensitive*.
6. Look at the response that comes back. Do you see the headers? They should look something like this:

   ```
   HTTP/1.1 200 OK
   Date: Tue, 28 Mar 2000 02:42:37 GMT
   Server: Apache/1.3.6 (Unix)
   Connection: close
   Content-Type: text/html
   ```

   This will be followed by a blank line, then the content of the file you asked for. Then you will see "Connection closed by foreign host", indicating that the HTTP server has closed the connection.

   If you miss seeing the headers because the body is too long, try using the `HEAD` method instead of `GET`.

7. Telnet to port 80 again and ask the web server for a CGI script's output by typing `GET /cgi-bin/localtime.cgi HTTP/1.0`
8. Now let's get some status codes other than `200 OK` from the web server:
   - `GET /not_here.html HTTP/1.0` (a file which doesn't exist)
   - `GET /unreadable.html HTTP/1.0` (a file with the permissions set wrong)
   - `GET /protected.html HTTP/1.0` (a file protected by HTTP authentication - we cover this later on today)

- `GET /redirected.html HTTP/1.0` (a file which is redirected to a different URL)
- `ENCRYPT /index.html HTTP/1.0` (a method which isn’t known to our server)

2.4. What is needed to run 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. The most common security requirement is that CGI programs must run under cgiwrap. This is discussed later, in the section on security.

2.5. 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

3.1. 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.

3.2. 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. So if your login name is `train03` and you are using the Netizen training server at `training.netizen.com.au`, you can access your web pages at `http://training.netizen.com.au/~train03`. Of course, you will need to replace both the hostname and username to match your specific setup.

3.3. 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.

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.
5. That all your CGI scripts are world readable and executable

3.4. The HTTP headers

Every CGI script must output an HTTP header giving a MIME content type, such as `Content-type: text/html`, with a blank line after it:

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

Put this at the top of every CGI script, as the first thing that’s printed.

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`

3.5. HTML output

Any other output of your script will be sent back to the web browser just as you specify. Since we're outputting content of the type `text/html` we should make our scripts output HTML:

```
print "<h1>Hello, world!</h1>\n";
```

The above example is already in your `cgi-bin` directory as `hello.cgi`.

3.6. Running and debugging 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 check how your program is running:

1. First, check that your program runs by running it from the command line. It may be that you've made a syntax error, or that your program has the wrong permissions
2. Second, try opening it in a browser. If your program runs on the command line but does not output content to the browser, you may have forgotten to print out the `Content-type: text/html` header, or forgotten to leave a blank line between the header and the body, or may have made an error in your HTML output.
3. Thirdly, check the web server's log files. Where these are will vary from system to system. On our system, they're in `/var/log/apache`, and you can check them using **cat**, **less**, **tail**, or any other tool of your choice. If you don't know what these commands do, check their manual pages by typing **man cat**, **man less**, etc.

3.6.1. Exercises

1. Look at the output of the `hello.cgi` script by pointing a web browser (such as Netscape) at `http://`*`hostname`*`/~train`*`XX`*`/cgi-bin/hello.cgi` (replace *`hostname`* with the hostname or IP address of the training server, and *`XX`* with your number)
2. Modify `hello.cgi` to set a variable `$name` and include that name in the greeting. (Don't forget to `use strict;`)
3. Run your modified `hello.cgi` from the command line to ensure that it runs.
4. Press the `Reload` button in your browser to see if your modifications worked correctly.

3.7. Quoting made easy

It can be annoying to output HTML using double quotes. You may find yourself doing things like this:

```
print "<img src=\"$img\" alt=\"$alttext\">\n";
print "<a href=\"$url\">A hypertext link</a>\n";
```

Escaping all those quotes with backslashes can get tedious and unreadable. Luckily, there are a couple of ways around it.

3.7.1. Here documents

"Here" documents allow you to print everything until a certain marker is found:

```
print <<"END";
<img src="$img" alt="$alttext">
<a href="$url">A hypertext link</a>
```

```
END
```

You can specify what end marker you want on in the `print` statement.

The fact that the marker is in double quotes means that the material up until the end marker is found will undergo interpolation in the same way as any double-quoted string. If you use single quotes, it'll act like a single-quoted string, and no interpolation will occur.

If you use backticks, it will execute each line via the shell.

The end marker must be on a line by itself, at the very start of the line. Note also that the `print` statement has a semi-colon on the end.

3.7.2. Pick your own quotes

Another way of avoiding excessive backslashes in your code is to use the `qq()` or `q()` operators/functions.

These quotes are covered on page 41 of the Camel book.

```
print qq(<img src="$img" alt="$alttext">\n);
print qq(<a href="$url">A hypertext link</a>\n);
```

Like the matching and substitution operators `m//` and `s///`, the quoting operators can use just about any character as a delimiter:

```
print qq(<a href="$url">A hypertext link</a>\n);
print qq!<a href="$url">A hypertext link</a>\n!;
print qq[<a href="$url">A hypertext link</a>\n];
print qq#<a href="$url">A hypertext link</a>\n#;
```

If the opening delimiter is a bracket type character, the closing delimiter will be the matching bracket.

Always choose a delimiter that isn't likely to be found in your quoted text. A slash, while common in non-HTML uses of the function, is not very useful for quoting anything containing HTML closing tags like `</p>`.

3.7.3. Exercises

The following exercises practice using CGI to output different Perl data types (as taught in Introduction to Perl) such as arrays and hashes. You may use plain double quotes, “here” documents, or the quoting operators as you see fit.

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. If you need help with HTML, there’s a cheat sheet in Appendix D.
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`

3.8. 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).

3.8.1. 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 an HTML page that points to your CGI program and see what the `REFERER` environment variable says.

3.9. Chapter summary

- CGI scripts are programs written in Perl or any other language that output web content such as HTML pages
- CGI scripts must output a Content-type header and a blank line before anything else

- Debugging techniques for CGI:
  - Run the script from the command line
  - Try opening it in the browser
  - Check the logs
- Various techniques are available for quoting text, including "here" documents and Perl quoting functions such as `qq()`.
- The `%ENV` special variable can be used to access environment variables via CGI scripts, including such variables as HTTP_USER_AGENT and HTTP_REFERER

Chapter 4. Accepting and processing form input

4.1. In this chapter...

CGI programs are often used to accept and process data from HTML forms. In this section, we take a quick look at HTML forms and use the `CGI` module to parse form data.

4.2. 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:

4.3. 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

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

Other attributes exist, but will not be used in this course.

4.4. Input fields

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

4.4.1. TEXT

A text input field `<INPUT TYPE="TEXT" NAME="email_address">`

4.4.2. CHECKBOX

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

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

4.4.3. SELECT

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

```
<SELECT NAME="hobbies">
    <OPTION VALUE="philately">Philately</OPTION>
    <OPTION VALUE="gardening">Gardening</OPTION>
    <OPTION VALUE="programming">Programming</OPTION>
    <OPTION VALUE="cookery">Cookery</OPTION>
    <OPTION VALUE="reading">Reading</OPTION>
    <OPTION VALUE="bushwalking">Bushwalking</OPTION>
</SELECT>
```

4.4.4. SUBMIT

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

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

4.5. The `CGI` module

4.5.1. What is a module?

A module is a collection of useful functions which you can use in your programs. They are written by Perl people worldwide, and distributed mostly through CPAN, the Comprehensive Perl Archive Network.

Perl modules save you heaps of time - by using a module, you save yourself from "reinventing the wheel". Perl modules also tend to save you from making silly mistakes again and again while you try to figure out how to do a given task.

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 "percent-encoded" format:

```
name=Kirrily&company=Netizen%20Pty.%20Ltd.
```

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 `CGI` module.

Although it is part of the current Perl distribution, the `CGI` module is not documented in the Camel book. Instead, type **perldoc CGI** to read about it.

4.5.2. Using the `CGI` module

To use the `CGI` module, simply put the statement `use CGI;` at the top of your script, thus:

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

use strict;
use CGI;
```

4.5.3. Accepting parameters with `CGI`

To accept form parameters into our CGI script as variables, we can say that we specifically want to use the `params` part of the CGI module:

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

use strict;
use CGI ’param’;
```

This provides us with a new subroutine, `param`, which we can use to extract the value of the HTML form’s fields.

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

use strict;
use CGI ’param’;

my $name = param(’name’);
print "Content-type: text/html\n\n";
print "Hello, $name!";
```

4.5.4. Debugging with the `CGI` module’s offline mode

When you run a CGI script from the command line, you will see a prompt 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
```

4.5.5. Exercises

1. Write a simple form to ask the user for their name. Type in the above script and see

if it works.

2. Add some fields to your form, including a checkbox and a drop down menu, and print out their values. What are the default true/false values for a checkbox?
3. What happens if you use the `SELECT MULTIPLE` 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` module. Print them out using a `foreach` loop, as in earlier exercises.

4.6. Practical Exercise: Data validation

Your trainer will now demonstrate and discuss the use of `CGI` 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 ’param’;

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

my @errors;

push (@errors, "Year must be numeric") if param(’year’) =~ /\D/;
push (@er-
rors, "You must fill in your name") if param(’name’) eq "";
push (@errors, "URL must be-
gin with http://") if param(’url’) !~ m!^http://!;

if (@errors) {
        print "<h2>Errors</h2>\n";
        print "<ul>\n";
        foreach (@errors) {
                print "<li>$_\n";
        }
        print "</ul>\n";
} else {
        print "<p>Congratulations, no errors!</p>\n";
}
```

4.6.1. Exercises

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

4.7. 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.

```
<INPUT TYPE="HIDDEN" VALUE="..." NAME="...">
```

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 ’param’;

print <<"END";
Content-type: text/html

<html>
<body>
<h1>Wizard interface</h1>
END

my $step = param(’step’) || 0;

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

print <<"END";
</body>
</html>
END
```

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

```
#
# Step 1 -- Name
#

sub step1 {
        print qq(
                <h2>Step 1: Name</h2>
                <p>
                What is your name?
                </p>
                <form method="POST" action="wizard.cgi">
                <input type="hidden" name="step" value="2">
                <input type="text" name="name">
                <input type="submit">
                </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 {
        my $name = param(’name’);
        print qq(
                <h2>Step 2: Quest</h2>
                <p>
                What is your quest?
                </p>
                <form method="POST" action="wizard.cgi">
                <input type="hidden" name="step" value="3">
                <input type="hidden" name="name" value="$name">
                <input type="text" name="quest">
                <input type="submit">
                </form>
        );
}


#
# Step 3 -- favourite colour
#

sub step3 {
        my $name = param(’name’);
        my $quest = param(’quest’);

        print qq(
                <h2>Step 3: Silly Question</h2>
                <p>
                What is the airspeed velocity of an un-
laden swallow?
                </p>
                <form method="POST" action="wizard.cgi">
                <input type="hidden" name="step" value="4">
                <input type="hidden" name="name" value="$name">
                <input type="hidden" name="quest" value="$quest">
                <input type="text" name="swallow">
                <input type="submit">
                </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 qq(
                <h2>Step 4: Done!</h2>
                <p>
                Thank you!
                </p>
                <p>
                Your name is $name.  Your quest is $quest.  The airspeed
                velocity of an unladen swallow is $swallow.
                </p>
        );
}
```

4.7.1. Exercises

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

4.8. Practical Exercise: File upload

`CGI` 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"`.

```
<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` handles file uploads quite easily. Just use `param()` as usual. 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 ’param’;

my $filename = param(’filename’);
my $outfile = "outputfile";

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

# There will probably be permission problems with this open
# statement unless you’re running under cgiwrap, or your script
# is setuid, or $outfile is world writable.  But let’s not worry
# about that for now.

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

# This bit is taken straight from the CGI.pm documentation --
# you could also just use "while (<$filename>)" if you wanted

my ($buffer, $bytesread);
while ($bytesread=read($filename,$buffer,1024)) {
        print OUTFILE $buffer;
}

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

print "<p>Uploaded file and saved as $outfile</p>\n";

print "</body></html>";
```

4.9. Chapter summary

- The `CGI` module can be used to parse data from HTML forms
- Its most common use is parameter parsing; other functions are also available
- To use it, type `use CGI 'param';` at the top of your script
- Obtain each item of data using the `param()` function
- `CGI` can be used to implement web applications of any complexity, including data validation, multi-form wizards, file upload, and more

Chapter 5. Security issues

5.1. In this chapter...

In this section we 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.

5.2. 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 necessarily 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.

5.2.1. 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' referer logs, and so on.
- Sometimes username and password details are passed back and forth using "cookies". Many 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

On the other hand, the main disadvantage of HTTP authentication is that the authentication tokens remain active until the user shuts their browser down. This can be a problem in public computer labs and other locations where users may share PCs.

5.2.2. 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 "Unauthorized", 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 `Authorization:` 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.

5.2.3. Access control

The way access control 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 `.htaccess` file in the web directory, which contains server configuration directives. Our server has been set up to allow you to use the `.htaccess` file.

A password file has already been set up for your use. It's `/etc/apache/training.passwd` and uses the same usernames and passwords as your login accounts. You can look at it by typing `cat /etc/apache/training.passwd`

To use this password file, create a file in your `public_html` directory called `.htaccess`, containing the following text:

```
AuthType Basic
AuthName "Secret stuff"
AuthUserFile /etc/apache/training.passwd
require valid-user
```

This authentication will apply to the directory in which the `.htaccess` file is placed and any subdirectories.

5.2.4. Exercises

1. Create a `.htaccess` file in your `public_html` directory, as above
2. Use your web browser to request one of your HTML files or CGI scripts, and observe the authentication process
3. Why would it be a bad idea to put the password file in the same directory as the web pages or CGI scripts?

5.3. 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 exe-
cute shell command
```

Imagine if the user’s input had been `skud; cat /etc/passwd`, or worse yet, `skud; 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 356 of the Camel.

`-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.

5.3.1. 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.

5.4. cgiwrap

Many large sites, such as ISPs and educational institutions, require users to run their CGI scripts using a program called **cgiwrap**. This program causes the CGI script to execute as if being run by the owner, instead of the web server's user ID. What this means is that the script will have permission to read and write the user's files, and will not be able to cause any damage on the system that the user could not cause.

5.5. 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 online credit card transactions more secure.

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

5.6. 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**.
- Some web servers use **cgiwrap** to run CGI scripts under their owner's user ID.

- Secure HTTP can be used to provide an encrypted channel of communication between the web client and server.

Chapter 6. Other related Perl modules

6.1. In this chapter...

In this section we are briefly introduced to Perl modules which may be useful to us in developing CGI applications, including modules for failing gracefully, encoding and decoding URLS, and filling in templates.

6.2. Useful Perl modules

There are several common problems faced by CGI programmers: failing gracefully, creating valid URLs from any text, using a template to insert variables into HTML, sending email based on CGI parameters, et cetera. Since these problems are so common, people have written modules to solve them. This section explains some of the most useful modules to save you from having to re-invent the wheel.

Each of these modules can be downloaded from CPAN (the Comprehensive Perl Archive Network) (http://www.perl.com/CPAN) and installed either using the CPAN module distributed with Perl, or by following the steps described in the `README` file distributed with each module.

6.3. 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, we can cause any fatal errors to have their error messages and diagnostic information output directly to the browser:

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

You may also like to look at the equivalent module for non-CGI scripts, the basic Carp module, which is documented on page 385 of your Camel book.

6.3.1. Exercise

1. Use the `CGI::Carp` module in one of your scripts
2. Deliberately cause a syntax error, for instance by removing a semi-colon or quote mark, or inserting a `die ("Argh!");` statement, and see what happens

6.4. Encoding URIs with `URI::Escape`

Sometimes we want to output anchor tags `<A HREF="...">` referring to another CGI script, and pass parameters along with it, thus:

```
<A HREF="lookup.cgi?title=Programming Perl&publisher=O'Reilly">
```

```
O’Reilly’s Programming Perl
</A>
```

However, spaces and apostrophes aren’t allowed in URIs, so we have to encode them into the "percent-encoded" format. This format replaces most non-alphanumeric characters with two hexadecimal digits. For instance, a space becomes `%20` and a tilde becomes `%7E`.

The Perl module we use to encode URIs in this manner is `URI::Escape`. Its documentation is available by typing **perldoc URI::Escape**.

Use it as follows:

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

use strict;
use URI::Escape;

my $book_lookup =
"lookup.cgi?title=Programming Perl&publisher=O’Reilly";

my $encoded_url = uri_escape($address);
my $original_url = uri_unescape($encoded_url);
```

6.4.1. Exercise

1. Try out the above script `cgi-bin/escape.cgi` you’ll need to print out the values of `$encoded_url` and `$original_url`

6.5. Creating templates with `Text::Template`

By this stage in the day you have probably spent a great deal of time outputting HTML either via a long list of `print` statements or by using a "here document" or other shortcut. What if you wanted to have a template HTML output file which was filled in with the appropriate variables?

Luckily, there is a Perl module to do this, called `Text::Template`. Unluckily, it uses a concept we haven't covered yet, but which we will now explain.

`Text::Template` is different to the other modules we have used so far today, in that it is an *object oriented* module. Object oriented Perl modules can be very powerful, but require some background knowledge to understand how they work.

6.5.1. Introduction to object oriented modules

Before embarking on this task, we need to have an understanding of how Perl's object-oriented modules work. Not all modules are object oriented (`URI::Escape`, for example, is not), and some can be used either way (CGI is one of these), but some require us to work with them in this way.

Read **perldoc CGI** if you want to know all about the OO interface to the `CGI` module, which provides many additional features.

A software object, like a real-life object, has attributes (things that describe the object) and methods (things you can do with, or to, the object). Consider the real-life example of a cup:

Table 6-1. Attributes and Methods of a cup

Object	Attributes	Methods

Object	**Attributes**	**Methods**
Cup	• colour • handle (does it have one?) • contents (water, coffee, etc) • fullness	• drink from it • fill it up • smash it

Note that when you smash a cup, you aren't smashing the generic class of cups, but rather a specific instance - *this* cup, not "cups in general". This is what we call an *instance of a class* -- remember that, as we'll use it later.

6.5.2. Using the `Text::Template` module

Like the cup, our text template has attributes and methods.

Table 6-2. Attributes and Methods of `Text::Template`

`Text::Template`	• `TYPE - the type of template it is, eg a file, a string you created earlier, etc` • `SOURCE - the filehandle or variable name in which the template can be found`	• `fill_in() - fill in the template`

Before we can actually use these attributes and methods in any useful way, we have to create a new instance of the class. This is the same as how we needed a specific cup, rather than the general class of cups.

```
# using the class in general
use Text::Template;

# instantiating the class and setting some at-
tributes for the new instance
my $let-
ter = new Text::Template{'TYPE' => 'FILE', 'SOURCE' => 'let-
ter.tmpl'};
```

We can then perform a method on it, thus:

```
my $finished_letter = $letter->fill_in();
```

This will fill in any variables found in the template file.

6.5.3. Exercise

1. Type **perldoc Text::Template** and look at the documentation for this module
2. `cgi-bin/letter.cgi` implements the example above. Examine the source code.
3. Make some changes to the letter template and see if they work.

6.6. Sending email with `Mail::Mailer`

The `Mail::Mailer` module can be used to send email from a CGI script (or, for that matter, any script). Like `Text::Template`, it is an Object Oriented module. The object it creates is a "mailer" object, which can be opened and then printed to as if it were a filehandle.

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

use strict;
use Mail::Mailer;

my $mailer = new Mail::Mailer;

# the open() method takes a hash refer-
ence with keys which are mail
# header names and values which are the val-
ues of those mail headers

$mailer->open( {
        From    =>      'fred@example.com',
        To      =>      'barney@example.com',
        Subject =>      'Web form submission'
} );

# we can print to $mailer just as we would print to STD-
OUT or any
# other file handle...

print $mailer qq(
Dear Barney,

Here is a form submission from your website:

Name:           $name
Email:          $email
Comments:       $comments

Love, Fred.

);

$mailer->close();
```

You can also open a pipe to **sendmail** directly, but doing this correctly can be difficult. This is why we recommend `Mail::Mailer` to avoid re-inventing the wheel.

6.6.1. Exercises

1. Create an HTML form with fields for name, email and comment
2. Use the above script (`cgi-bin/mail.cgi`) to mail the results of the script to yourself. You will need to edit it to work fully:
   - Use CGI.pm to pick up the parameters
   - Change the email address to your own address
   - Print out a "thank you" page once the form has been submitted -- don't forget the Content-type header

6.7. Chapter Summary

- The `CGI::Carp` module can be used to help CGI programs fail gracefully
- The `URI::Escape` module can be used to encode and decode percent-encoded URLs
- The `Text::Template` module can be used to easily fill in text templates, including HTML templates.
- The `Mail::Mailer` module can be used to send email based on the information entered in an HTML form
- All these modules can be downloaded from CPAN, the Comprehensive Perl Archive Network

Chapter 7. Conclusion

7.1. What you’ve learnt

Now you’ve completed Netizen’s 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 retreive documents
- How to perform HTTP requests by using **telnet** to connect to the web server
- How to generate simple web pages using Perl
- How to access environment variables from CGI scripts
- Various methods of quoting text, including "here" documents and the `qq()` type functions
- 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.
- The use of various Perl modules related to CGI programming, including CGI::Carp, URI::Escape, Text::Template, and Mail::Mailer
- A basic understanding of object oriented Perl modules

7.2. Where to now?

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

- Borrow or purchase the books listed 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/)
- Extend your knowledge with further Netizen courses such as:
  - Web enabled databases with Perl and DBI

  Information about these courses can be found on Netizen's website (http://netizen.com.au/services/training/). A diagram of Netizen's courses and the careers they can lead to is included with these training materials.

7.3. Further reading

- 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/)

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.

B.1. vi

B.1.1. Running

```
% vi filename
```

B.1.2. 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.

B.1.3. 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.

B.1.4. 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).

B.1.5. Help

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

B.2. pico

B.2.1. Running

```
% pico -w filename
```

B.2.2. 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.

B.2.3. Exiting

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

B.2.4. 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. \\ \hline

B.2.5. Help

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

B.3. joe

B.3.1. Running

```
% joe filename
```

B.3.2. Using

- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-K** then **S** to save.

B.3.3. Exiting

- **CTRL-C** to exit without save.
- **CTRL-K** then **X** to save and exit.

B.3.4. Gotchas

Nothing in particular.

B.3.5. Help

CTRL-K then **H**.

B.4. jed

B.4.1. Running

```
% jed
```

B.4.2. Using

- Defaults to the emacs emulation mode.
- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-X** then **S** to save.

B.4.3. Exiting

CTRL-X then **CTRL-C** to exit.

B.4.4. Gotchas

Nothing in particular.

B.4.5. Help

- Read the menu bar at the top.
- Press **ESC** then **?** then **H** from the main menu.

Appendix C. ASCII Pronunciation Guide

Table C-1. ASCII Pronunciation Guide

Character	Pronunciation
!	bang, exlamation
*	star, asterisk
$	dollar
@	at
%	percent
&	ampersand
"	double-quote
’	single-quote, tick
( )	open/close bracket, parentheses
<	less than
>	greater than
-	dash, hyphen
.	dot
,	comma
/	slash, forward-slash
\	backslash, slosh
:	colon
;	semi-colon
=	equals
?	question-mark
^	caret (pron. carrot)
_	underscore

Character	**Pronunciation**
[ ]	open/close square bracket
{ }	open/close curly brackets, open/close brace
|	pipe, or vertical bar
~	tilde (pron. “til-duh”, 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>`

Type of information	**Markup**
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/)