**http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html**

**CGI.pm - a Perl5 CGI Library**

Version 2.98, 7/15/2003, L. Stein

**Abstract**

This perl 5 library uses objects to create Web fill-out forms on the fly and to parse their contents. It provides a simple interface for parsing and interpreting query strings passed to CGI scripts. However, it also offers a rich set of functions for creating fill-out forms. Instead of remembering the syntax for HTML form elements, you just make a series of perl function calls. An important fringe benefit of this is that the value of the previous query is used to initialize the form, so that the state of the form is preserved from invocation to invocation.

Everything is done through a ``CGI'' object. When you create one of these objects it examines the environment for a query string, parses it, and stores the results. You can then ask the CGI object to return or modify the query values. CGI objects handle POST and GET methods correctly, and correctly distinguish between scripts called from <ISINDEX> documents and form-based documents. In fact you can debug your script from the command line without worrying about setting up environment variables.

A script to create a fill-out form that remembers its state each time it's invoked is very easy to write with CGI.pm:

#!/usr/local/bin/perl

use CGI qw(:standard);

print header;

print start\_html('A Simple Example'),

h1('A Simple Example'),

start\_form,

"What's your name? ",textfield('name'),

p,

"What's the combination?",

p,

checkbox\_group(-name=>'words',

-values=>['eenie','meenie','minie','moe'],

-defaults=>['eenie','minie']),

p,

"What's your favorite color? ",

popup\_menu(-name=>'color',

-values=>['red','green','blue','chartreuse']),

p,

submit,

end\_form,

hr;

if (param()) {

print

"Your name is",em(param('name')),

p,

"The keywords are: ",em(join(", ",param('words'))),

p,

"Your favorite color is ",em(param('color')),

hr;

}

print end\_html;

[Select this link to try the script](http://pages.cs.wisc.edu/~beechung/ref/examples/tryit.cgi)  
[More scripting examples](http://pages.cs.wisc.edu/~beechung/ref/examples/)  
[Source code examples from *The Official Guide to CGI.pm*](http://www.wiley.com/compbooks/stein/source.html)

**Contents**

 [Downloading](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#download)

 [Installation](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#installation)

 [Function-Oriented vs Object-Oriented Use](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#functionvsoo)

 [Creating a new CGI query object](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#query)

 [Saving the state of the form](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#saving)

 [CGI Functions that Take Multiple Arguments](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#named_param)

 [Creating the HTTP header](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#header)

 [HTML shortcuts](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#html)

 [Creating forms](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#forms)

 [Importing CGI methods](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#import)

 [Retrieving CGI.pm errors](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#errors)

 [Debugging](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#debugging)

 [HTTP session variables](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#environment)

 [HTTP Cookies](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#cookies)

 [Support for frames](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#frames)

 [Support for JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting)

 [Limited Support for Cascading Style Sheets](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#stylesheets)

 [Using NPH Scripts](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#nph)

 [Advanced techniques](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#advanced)

 [Subclassing CGI.pm](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#subclassing)

 [Using CGI.pm with mod\_perl and FastCGI](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#mod_perl)

 [Migrating from cgi-lib.pl](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#migrating)

 [Using the File Upload Feature](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#upload_caveats)

 [Server Push](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#push)

 [Avoiding Denial of Service Attacks](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#dos)

 [Using CGI.pm on non-Unix Platforms](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#non_unix)

 [The Relationship of CGI.pm to the CGI::\* Modules](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#future)

 [Distribution information](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#distribution)

 [The CGI.pm Book](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#book)

 [CGI.pm and the Year 2000 Problem](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#y2000)

 [Bug Reporting and Support](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#bugs)

 [What's new?](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#new)

**Downloads**

* [**Download gzip tar archive (Unix)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.tar.gz)
* [**Download pkzip archive (Windows)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.zip)
* [**Download sit archive (Macintosh)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.sit)
* [**Download just the CGI module (uncompressed)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm)
* [**Archive of Old Versions**](http://pages.cs.wisc.edu/~beechung/ref/old)

**Installation**

* [**Download gzip tar archive (Unix)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.tar.gz)
* [**Download pkzip archive (Windows)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.zip)
* [**Download sit archive (Macintosh)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.sit)
* [**Download just the CGI module (uncompressed)**](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm)

The current version of the software can always be downloaded from the master copy of this document maintained at <http://stein.cshl.org/WWW/software/CGI/>.

This package requires perl 5.004 or higher. Earlier versions of Perl may work, but CGI.pm has not been tested with them. If you're really stuck, edit the source code to remove the line that says "require 5.004", but don't be surprised if you run into problems.

If you are using a Unix system, you should have perl do the installation for you. Move to the directory containing CGI.pm and type the following commands:

% perl Makefile.PL

% make

% make install

You may need to be root to do the last step.

This will create two new files in your Perl library. **CGI.pm** is the main library file. **Carp.pm** (in the subdirectory "CGI") contains some optional utility routines for writing nicely formatted error messages into your server logs. See the Carp.pm man page for more details.

**If you get error messages when you try to install**, then you are either:

1. Running a Windows NT or Macintosh port of Perl that doesn't have make or the MakeMaker program built into it.
2. Have an old version of Perl. Upgrade to 5.004 or higher.

In the former case don't panic. Here's a recipe that will work (commands are given in MS-DOS/Windows form):

> cd CGI.pm-2.73

> copy CGI.pm C:\Perl\lib

> mkdir C:\Perl\lib\CGI

> copy CGI\\*.pm C:\Perl\lib\CGI

Modify this recipe if your Perl library has a different location.

For Macintosh users, just drag the file named CGI.pm into the folder where your other Perl .pm files are stored. Also drag the subfolder named "CGI".

**If you do not have sufficient privileges to install into /usr/local/lib/perl5**, you can still use CGI.pm. Modify the installation recipe as follows:

% perl Makefile.PL INSTALLDIRS=site INSTALLSITELIB=/home/your/private/dir

% make

% make install

Replace */home/your/private/dir* with the full path to the directory you want the library placed in. Now preface your CGI scripts with a preamble something like the following:

use lib '/home/your/private/dir';

use CGI;

Be sure to replace /home/your/private/dir with the true location of CGI.pm.

[Notes on using CGI.pm in NT and other non-Unix platforms](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#non_unix)

**Function-Oriented vs Object-Oriented Use**

CGI.pm can be used in two distinct modes called *function-oriented* and *object-oriented*. In the function-oriented mode, you first import CGI functions into your script's namespace, then call these functions directly. A simple function-oriented script looks like this:

#!/usr/local/bin/perl

use CGI qw/:standard/;

print header(),

start\_html(-title=>'Wow!'),

h1('Wow!'),

'Look Ma, no hands!',

end\_html();

The *use* operator loads the CGI.pm definitions and imports the ":standard" set of function definitions. We then make calls to various functions such as *header()*, to generate the HTTP header, *start\_html()*, to produce the top part of an HTML document, *h1()* to produce a level one header, and so forth.

In addition to the standard set, there are many optional sets of less frequently used CGI functions. See [Importing CGI Methods](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#import) for full details.

In the object-oriented mode, you *use CGI;* without specifying any functions or function sets to import. In this case, you communicate with CGI.pm via a CGI object. The object is created by a call to *CGI::new()* and encapsulates all the state information about the current CGI transaction, such as values of the CGI parameters passed to your script. Although more verbose, this coding style has the advantage of allowing you to create multiple CGI objects, save their state to disk or to a database, and otherwise manipulate them to achieve neat effects.

The same script written using the object-oriented style looks like this:

#!/usr/local/bin/perl

use CGI;

$q = new CGI;

print $q->header(),

$q->start\_html(-title=>'Wow!'),

$q->h1('Wow!'),

'Look Ma, no hands!',

$q->end\_html();

The object-oriented mode also has the advantage of consuming somewhat less memory than the function-oriented coding style. This may be of value to users of persistent Perl interpreters such as [mod\_perl](http://perl.apache.org/).

Many of the code examples below show the object-oriented coding style. Mentally translate them into the function-oriented style if you prefer.

**Creating a new CGI object**

The most basic use of CGI.pm is to get at the query parameters submitted to your script. To create a new CGI object that contains the parameters passed to your script, put the following at the top of your perl CGI programs:

use CGI;

$query = new CGI;

In the object-oriented world of Perl 5, this code calls the new() method of the CGI class and stores a new CGI object into the variable named $query. The new() method does all the dirty work of parsing the script parameters and environment variables and stores its results in the new object. You'll now make method calls with this object to get at the parameters, generate form elements, and do other useful things.

An alternative form of the new() method allows you to read script parameters from a previously-opened file handle:

$query = new CGI(FILEHANDLE)

The filehandle can contain a URL-encoded query string, or can be a series of newline delimited TAG=VALUE pairs. This is compatible with the save() method. This lets you save the state of a CGI script to a file and reload it later. It's also possible to save the contents of several query objects to the same file, either within a single script or over a period of time. You can then reload the multiple records into an array of query objects with something like this:

open (IN,"test.in") || die;

while (!eof(IN)) {

my $q = new CGI(IN);

push(@queries,$q);

}

You can make simple databases this way, or create a guestbook. If you're a Perl purist, you can pass a reference to the filehandle glob instead of the filehandle name. This is the "official" way to pass filehandles in Perl5:

my $q = new CGI(\\*IN);

(If you don't know what I'm talking about, then you're not a Perl purist and you needn't worry about it.)

If you are using the function-oriented interface and want to initialize CGI state from a file handle, the way to do this is with *restore\_parameters()*. This will (re)initialize the default CGI object from the indicated file handle.

open (IN,"test.in") || die;

restore\_parameters(IN);

close IN;

You can initialize a CGI object from an associative-array reference. Values can be either single- or multivalued:

$query = new CGI({'dinosaur'=>'barney',

'song'=>'I love you',

'friends'=>[qw/Jessica George Nancy/]});

You can initialize a CGI object by passing a URL-style query string to the new() method like this:

$query = new CGI('dinosaur=barney&color=purple');

Or you can clone a CGI object from an existing one. The parameter lists of the clone will be identical, but other fields, such as autoescaping, are not:

$old\_query = new CGI;

$new\_query = new CGI($old\_query);

This form also allows you to create a CGI object that is initially empty:

$empty\_query = new CGI('');

If you are using mod\_perl, you can initialize a CGI object at any stage of the request by passing the request object to CGI->new:

$q = CGI->new($r);

To do this with the function-oriented interface, set Apache->request($r) before calling the first CGI function.

See [advanced techniques](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#advanced) for more information.

**Fetching A List Of Keywords From The Query**

@keywords = $query->keywords

If the script was invoked as the result of an <ISINDEX> search, the parsed keywords can be obtained with the keywords() method. This method will return the keywords as a perl array.

**Fetching The Names Of All The Parameters Passed To Your Script**

@names = $query->param

If the script was invoked with a parameter list (e.g. "name1=value1&name2=value2&name3=value3"), the param() method will return the parameter names as a list. For backwards compatibility if the script was invoked as an <ISINDEX> script and contains a string without ampersands (e.g. "value1+value2+value3") , there will be a single parameter named "keywords" containing the "+"-delimited keywords.

**Fetching The Value(s) Of A Named Parameter**

@values = $query->param('foo');

-or-

$value = $query->param('foo');

Pass the param() method a single argument to fetch the value of the named parameter. If the parameter is multivalued (e.g. from multiple selections in a scrolling list), you can ask to receive an array. Otherwise the method will return a single value.

If a value is not given in the query string, as in the queries "name1=&name2=" or "name1&name2", it will be returned as an empty string (not undef). This feature is new in 2.63, and was introduced to avoid multiple "undefined value" warnings when running with the -w switch.

If the parameter does not exist at all, then param() will return undef in a scalar context, and the empty list in a list context.

**Setting The Value(s) Of A Named Parameter**

$query->param('foo','an','array','of','values');

-or-

$query->param(-name=>'foo',-values=>['an','array','of','values']);

This sets the value for the named parameter 'foo' to one or more values. These values will be used to initialize form elements, if you so desire. Note that this is the one way to forcibly change the value of a form field after it has previously been set.

The second example shows an alternative "named parameter" style of function call that is accepted by most of the CGI methods. See [Calling CGI functions that Take Multiple Arguments](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#named_param) for an explanation of this style.

**Appending a Parameter**

$query->append(-name=>'foo',-values=>['yet','more','values']);

This adds a value or list of values to the named parameter. The values are appended to the end of the parameter if it already exists. Otherwise the parameter is created.

**Deleting a Named Parameter Entirely**

$query->delete('foo');

This deletes a named parameter entirely. This is useful when you want to reset the value of the parameter so that it isn't passed down between invocations of the script.

**Deleting all Parameters**

$query->delete\_all();

This deletes all the parameters and leaves you with an empty CGI object. This may be useful to restore all the defaults produced by the form element generating methods.

**Handling non-URLencoded Arguments**

If POSTed data is not of type application/x-www-form-urlencoded or multipart/form-data, then the POSTed data will not be processed, but instead be returned as-is in a parameter named POSTDATA. To retrieve it, use code like this:

my $data = $query->param('POSTDATA');

(If you don't know what the preceding means, don't worry about it. It only affects people trying to use CGI for XML processing and other specialized tasks.)

**Importing parameters into a namespace**

$query->import\_names('R');

print "Your name is $R::name\n"

print "Your favorite colors are @R::colors\n";

This imports all parameters into the given name space. For example, if there were parameters named 'foo1', 'foo2' and 'foo3', after executing $query->import\_names('R'), the variables @R::foo1, $R::foo1, @R::foo2, $R::foo2, etc. would conveniently spring into existence. Since CGI has no way of knowing whether you expect a multi- or single-valued parameter, it creates two variables for each parameter. One is an array, and contains all the values, and the other is a scalar containing the first member of the array. Use whichever one is appropriate. For keyword (a+b+c+d) lists, the variable @R::keywords will be created.

If you don't specify a name space, this method assumes namespace "Q".

An optional second argument to **import\_names**, if present and non-zero, will delete the contents of the namespace before loading it. This may be useful for environments like mod\_perl in which the script does not exit after processing a request.

**Warning**: do not import into namespace 'main'. This represents a major security risk, as evil people could then use this feature to redefine central variables such as @INC. CGI.pm will exit with an error if you try to do this.

**NOTE:** Variable names are transformed as necessary into legal Perl variable names. All non-legal characters are transformed into underscores. If you need to keep the original names, you should use the param() method instead to access CGI variables by name.

**Direct Access to the Parameter List**

$q->param\_fetch('address')->[1] = '1313 Mockingbird Lane';

unshift @{$q->param\_fetch(-name=>'address')},'George Munster';

If you need access to the parameter list in a way that isn't covered by the methods above, you can obtain a direct reference to it by calling the **param\_fetch()** method with the name of the parameter you want. This will return an array reference to the named parameters, which you then can manipulate in any way you like.

You may call **param\_fetch()** with the name of the CGI parameter, or with the **-name** argument, which has the same meaning as elsewhere.

**Fetching the Parameter List as a Hash**

$params = $q->Vars;

print $params->{'address'};

@foo = split("\0",$params->{'foo'});

%params = $q->Vars;

use CGI ':cgi-lib';

$params = Vars;

Many people want to fetch the entire parameter list as a hash in which the keys are the names of the CGI parameters, and the values are the parameters' values. The **Vars()** method does this. Called in a scalar context, it returns the parameter list as a tied hash reference. Changing a key changes the value of the parameter in the underlying CGI parameter list. Called in an array context, it returns the parameter list as an ordinary hash. This allows you to read the contents of the parameter list, but not to change it.

When using this, the thing you must watch out for are multivalued CGI parameters. Because a hash cannot distinguish between scalar and array context, multivalued parameters will be returned as a packed string, separated by the "\0" (null) character. You must split this packed string in order to get at the individual values. This is the convention introduced long ago by Steve Brenner in his cgi-lib.pl module for Perl version 4.

If you wish to use **Vars()** as a function, import the *:cgi-lib* set of function calls (also see the section on [CGI-LIB compatibility](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#migrating)).

**RETRIEVING CGI ERRORS**

Errors can occur while processing user input, particularly when processing uploaded files. When these errors occur, CGI will stop processing and return an empty parameter list. You can test for the existence and nature of errors using the **cgi\_error()** function. The error messages are formatted as HTTP status codes. You can either incorporate the error text into an HTML page, or use it as the value of the HTTP status:

my $error = $q->cgi\_error;

if ($error) {

print $q->header(-status=>$error),

$q->start\_html('Problems'),

$q->h2('Request not processed'),

$q->strong($error);

exit 0;

}

When using the function-oriented interface (see the next section), errors may only occur the first time you call **param()**. Be prepared for this! [Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Saving the Current State of a Form**

**Saving the State to a File**

$query->save(FILEHANDLE)

This writes the current query out to the file handle of your choice. The file handle must already be open and be writable, but other than that it can point to a file, a socket, a pipe, or whatever. The contents of the form are written out as TAG=VALUE pairs, which can be reloaded with the new() method at some later time. You can write out multiple queries to the same file and later read them into query objects one by one.

If you wish to use this method from the function-oriented (non-OO) interface, the exported name for this method is *save\_parameters()*. See [advanced techniques](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#advanced) for more information.

**Saving the State in a Self-Referencing URL**

$my\_url=$query->self\_url

This call returns a URL that, when selected, reinvokes this script with all its state information intact. This is most useful when you want to jump around within a script-generated document using internal anchors, but don't want to disrupt the current contents of the form(s). See [advanced techniques](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#advanced) for an example.

If you'd like to get the URL without the entire query string appended to it, use the url() method:

$my\_self=$query->url

**Obtaining the Script's URL**

$full\_url = $query->url();

$full\_url = $query->url(-full=>1); #alternative syntax

$relative\_url = $query->url(-relative=>1);

$absolute\_url = $query->url(-absolute=>1);

$url\_with\_path = $query->url(-path\_info=>1);

$url\_with\_path\_and\_query = $query->url(-path\_info=>1,-query=>1);

url() returns the script's URL in a variety of formats. Called without any arguments, it returns the full form of the URL, including host name and port number

http://your.host.com/path/to/script.cgi

You can modify this format with the following named arguments:

**-absolute**

If true, produce an absolute URL, e.g.

/path/to/script.cgi

**-relative**

Produce a relative URL. This is useful if you want to reinvoke your script with different parameters. For example:

script.cgi

**-full**

Produce the full URL, exactly as if called without any arguments. This overrides the -relative and -absolute arguments.

**-path**,**-path\_info**

Append the additional path information to the URL. This can be combined with -full, -absolute or -relative. -path\_info is provided as a synonym.

**-query** (**-query\_string**)

Append the query string to the URL. This can be combined with -full, -absolute or -relative. -query\_string is provided as a synonym.

**Mixing POST and URL Parameters**

$color = $query->url\_param('color');

It is possible for a script to receive CGI parameters in the URL as well as in the fill-out form by creating a form that POSTs to a URL containing a query string (a "?" mark followed by arguments). The **param()** method will always return the contents of the POSTed fill-out form, ignoring the URL's query string. To retrieve URL parameters, call the **url\_param()** method. Use it in the same way as **param()**. The main difference is that it allows you to read the parameters, but not set them.

Under no circumstances will the contents of the URL query string interfere with similarly-named CGI parameters in POSTed forms. If you try to mix a URL query string with a form submitted with the GET method, the results will not be what you expect.

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Calling CGI Functions that Take Multiple Arguments**

In versions of CGI.pm prior to 2.0, it could get difficult to remember the proper order of arguments in CGI function calls that accepted five or six different arguments. As of 2.0, there's a better way to pass arguments to the various CGI functions. In this style, you pass a series of name=>argument pairs, like this:

$field = $query->radio\_group(-name=>'OS',

-values=>[Unix,Windows,Macintosh],

-default=>'Unix');

The advantages of this style are that you don't have to remember the exact order of the arguments, and if you leave out a parameter, it will usually default to some reasonable value. If you provide a parameter that the method doesn't recognize, it will usually do something useful with it, such as incorporating it into the HTML tag as an attribute. For example if Netscape decides next week to add a new JUSTIFICATION parameter to the text field tags, you can start using the feature without waiting for a new version of CGI.pm:

$field = $query->textfield(-name=>'State',

-default=>'gaseous',

-justification=>'RIGHT');

This will result in an HTML tag that looks like this:

<INPUT TYPE="textfield" NAME="State" VALUE="gaseous"

JUSTIFICATION="RIGHT">

Parameter names are case insensitive: you can use -name, or -Name or -NAME. Actually, CGI.pm only looks for a hyphen in the first parameter. So you can leave it off subsequent parameters if you like. Something to be wary of is the potential that a string constant like "values" will collide with a keyword (and in fact it does!) While Perl usually figures out when you're referring to a function and when you're referring to a string, you probably should put quotation marks around all string constants just to play it safe.

HTML/HTTP parameters that contain internal hyphens, such as *-Content-language* can be passed by putting quotes around them, or by using an underscore for the second hyphen, e.g. *-Content\_language*.

The fact that you must use curly {} braces around the attributes passed to functions that create simple HTML tags but don't use them around the arguments passed to all other functions has many people, including myself, confused. As of 2.37b7, the syntax is extended to allow you to use curly braces for all function calls:

$field = $query->radio\_group({-name=>'OS',

-values=>[Unix,Windows,Macintosh],

-default=>'Unix'});

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Creating the HTTP Header**

**Creating the Standard Header for a Virtual Document**

print $query->header('image/gif');

This prints out the required HTTP Content-type: header and the requisite blank line beneath it. If no parameter is specified, it will default to 'text/html'.

An extended form of this method allows you to specify a status code and a message to pass back to the browser:

print $query->header(-type=>'image/gif',

-status=>'204 No Response');

This presents the browser with a status code of 204 (No response). Properly-behaved browsers will take no action, simply remaining on the current page. (This is appropriate for a script that does some processing but doesn't need to display any results, or for a script called when a user clicks on an empty part of a clickable image map.)

Several other named parameters are recognized. Here's a contrived example that uses them all:

print $query->header(-type=>'image/gif',

-status=>'402 Payment Required',

-expires=>'+3d',

-cookie=>$my\_cookie,

-charset=>'UTF-7',

-attachment=>'foo.gif',

-Cost=>'$0.02');

**-expires**

Some browsers, such as Internet Explorer, cache the output of CGI scripts. Others, such as Netscape Navigator do not. This leads to annoying and inconsistent behavior when going from one browser to another. You can force the behavior to be consistent by using the **-expires** parameter. When you specify an absolute or relative expiration interval with this parameter, browsers and proxy servers will cache the script's output until the indicated expiration date. The following forms are all valid for the **-expires** field:

+30s 30 seconds from now

+10m ten minutes from now

+1h one hour from now

-1d yesterday (i.e. "ASAP!")

now immediately

+3M in three months

+10y in ten years time

Thu, 25-Apr-1999 00:40:33 GMT at the indicated time & date

When you use **-expires**, the script also generates a correct time stamp for the generated document to ensure that your clock and the browser's clock agree. This allows you to create documents that are reliably cached for short periods of time.

**CGI::expires()** is the static function call used internally that turns relative time intervals into HTTP dates. You can call it directly if you wish.

**-cookie**

The **-cookie** parameter generates a header that tells Netscape browsers to return a "magic cookie" during all subsequent transactions with your script. HTTP cookies have a special format that includes interesting attributes such as expiration time. Use the [cookie()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#cookies) method to create and retrieve session cookies. The value of this parameter can be either a scalar value or an array reference. You can use the latter to generate multiple cookies. (You can use the alias **-cookies** for readability.)

**-nph**

The **-nph** parameter, if set to a non-zero value, will generate a valid header for use in no-parsed-header scripts. For example:

print $query->header(-nph=>1,

-status=>'200 OK',

-type=>'text/html');

You will need to use this if:

1. You are using Microsoft Internet Information Server.
2. If you need to create unbuffered output, for example for use in a "server push" script.
3. To take advantage of HTTP extensions not supported by your server.

See [Using NPH Scripts](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#nph) for more information.

**-charset**

The **-charset** parameter can be used to control the character set sent to the browser. If not provided, defaults to ISO-8859-1. As a side effect, this calls the charset() method to set the behavior for escapeHTML().

**-attachment**

The **-attachment** parameter can be used to turn the page into an attachment. Instead of displaying the page, some browsers will prompt the user to save it to disk. The value of the argument is the suggested name for the saved file. In order for this to work, you may have to set the **-type** to "application/octet-stream".

**-p3p**

The **-p3p** parameter will add a P3P tag to the outgoing header. The parameter can be an arrayref or a space-delimited string of P3P tags. For example:

print header(-p3p=>[qw(CAO DSP LAW CURa)]);

print header(-p3p=>'CAO DSP LAW CURa');

In either case, the outgoing header will be formatted as:

P3P: policyref="/w3c/p3p.xml" cp="CAO DSP LAW CURa"

**Other header fields**

Any other parameters that you pass to **header()** will be turned into correctly formatted HTTP header fields, even if they aren't called for in the current HTTP spec. For example, the example that appears a few paragraphs above creates a field that looks like this:

Cost: $0.02

You can use this to take advantage of new HTTP header fields without waiting for the next release of CGI.pm.

**Creating the Header for a Redirection Request**

print $query->redirect('http://somewhere.else/in/the/world');

This generates a redirection request for the remote browser. It will immediately go to the indicated URL. You should exit soon after this. Nothing else will be displayed.

You can add your own headers to this as in the header() method.

You should always use absolute or full URLs in redirection requests. Relative URLs will **not** work correctly.

An alternative syntax for redirect() is:

print $query->redirect(-location=>'http://somewhere.else/',

-nph=>1);

The **-location** parameter gives the destination URL. You may also use **-uri** or **-url** if you prefer.

The **-nph** parameter, if non-zero tells CGI.pm that this script is running as a no-parsed-header script. See [Using NPH Scripts](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#nph) for more information.

The **-method** parameter tells the browser what method to use for redirection. This is handy if, for example, your script was called from a fill-out form POST operation, but you want to redirect the browser to a static page that requires a GET.

All other parameters recognized by the header() method are also valid in redirect. [Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**HTML Shortcuts**

**Creating an HTML Header**

*named parameter style*

print $query->start\_html(-title=>'Secrets of the Pyramids',

-author=>'fred@capricorn.org',

-base=>'true',

-meta=>{'keywords'=>'pharoah secret mummy',

'copyright'=>'copyright 1996 King Tut'},

-style=>{'src'=>'/styles/style1.css'},

-dtd=>1,

-BGCOLOR=>'blue');

*old style*

print $query->start\_html('Secrets of the Pyramids',

'fred@capricorn.org','true');

This will return a canned HTML header and the opening <BODY> tag. All parameters are optional:

* The title (**-title**)
* The author's e-mail address (will create a <LINK REV="MADE"> tag if present (**-author**)
* A true flag if you want to include a <BASE> tag in the header (**-base**). This helps resolve relative addresses to absolute ones when the document is moved, but makes the document hierarchy non-portable. Use with care!
* A **-xbase** parameter, if you want to include a <BASE> tag that points to some external location. Example:
* print $query->start\_html(-title=>'Secrets of the Pyramids',
* -xbase=>'http://www.nile.eg/pyramid.html');

* A **-target** parameter, if you want to have all links and fill out forms on the page go to a different frame. Example:
* print $query->start\_html(-title=>'Secrets of the Pyramids',
* -target=>'answer\_frame');

**-target** can be used with either **-xbase** or **-base**.

* A **-meta** parameter to define one or more <META> tags. Pass this parameter a reference to an associative array containing key/value pairs. Each pair becomes a <META> tag in a format similar to this one.
* <META NAME="keywords" CONTENT="pharoah secret mummy">
* <META NAME="description" CONTENT="copyright 1996 King Tut">

To create an HTTP-EQUIV tag, use the **-head** argument as described below.

* The **-encoding** argument can be used to specify the character set for XHTML. It defaults to iso-8859-1 if not specified.
* A **-lang**> argument is used to incorporate a language attribute into the <HTM>> tag. The default if not specified is "en-US" for US English. For example:
* print $q->start\_html(-lang=>'fr-CA');

To leave off the lang attribute, as you must do if you want to generate legal HTML 3.2 or earlier, pass the empty string (-lang=>'').

* A **-dtd** parameter to make start\_html() generate an SGML document type definition for the document. This is used by SGML editors and high-end Web publishing systems to determine the type of the document. However, it breaks some browsers, in particular AOL's. The value of this parameter can be one of:
  1. A valid DTD (see [http://ugweb.cs.ualberta.ca/%7egerald/validate/lib/catalog](http://ugweb.cs.ualberta.ca/~gerald/validate/lib/catalog) for a list). Example:

-dtd=>'-//W3C//DTD HTML 3.2//EN'

* 1. A true value that does not begin with "-//", in which case you will get the standard default DTD (valid for HTML 2.0).

You can change the default DTD by calling **default\_dtd()** with the preferred value.

* A **-style** parameter to define a cascading stylesheet. More information on this can be found in [Limited Support for Cascading Style Sheets](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#stylesheets)
* A **-head** parameter to define other arbitrary elements of the <HEAD> section. For example:
* print start\_html(-head=>Link({-rel=>'next',
* -href=>'http://www.capricorn.com/s2.html'}));

or even

print start\_html(-head=>[ Link({-rel=>'next',

-href=>'http://www.capricorn.com/s2.html'}),

Link({-rel=>'previous',

-href=>'http://www.capricorn.com/s1.html'})

]

);

To create an HTTP-EQUIV tag, use something like this:

print start\_html(-head=>meta({-http\_equiv=>'Content-Type',

-content=>'text/html'}))

* A **-script** parameter to define Netscape [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) functions to incorporate into the HTML page. This is the preferred way to define a library of JavaScript functions that will be called from elsewhere within the page. CGI.pm will attempt to format the JavaScript code in such a way that non-Netscape browsers won't try to display the JavaScript code. Unfortunately some browsers get confused nevertheless. Here's an example of how to create a JavaScript library and incorporating it into the HTML code header:
* $query = new CGI;
* print $query->header;
* $JSCRIPT=<<END;
* // Ask a silly question
* function riddle\_me\_this() {
* var r = prompt("What walks on four legs in the morning, " +
* "two legs in the afternoon, " +
* "and three legs in the evening?");
* response(r);
* }
* // Get a silly answer
* function response(answer) {
* if (answer == "man")
* alert("Right you are!");
* else
* alert("Wrong! Guess again.");
* }
* END
* print $query->start\_html(-title=>'The Riddle of the Sphinx',
* -script=>$JSCRIPT);

Netscape 3.0 and higher allows you to place the JavaScript code in an external document and refer to it by URL. This allows you to keep the JavaScript code in a file or CGI script rather than cluttering up each page with the source. Netscape 3.X-4.X and Internet Explorer 3.X-4.X also recognize a "language" parameter that allows you to use other languages, such as VBScript and PerlScript (yes indeed!) To use these attributes pass a HASH reference in the **-script** parameter containing one or more of the keys **language**, **src**, or **code**. Here's how to refer to an external script URL:

print $q->start\_html(-title=>'The Riddle of the Sphinx',

-script=>{-language=>'JavaScript',

-src=>'/javascript/sphinx.js'}

);

Here's how to refer to scripting code incorporated directly into the page:

print $q->start\_html(-title=>'The Riddle of the Sphinx',

-script=>{-language=>'PerlScript',

-code=>'print "hello world!\n;"'}

);

A final feature allows you to incorporate multiple <SCRIPT> sections into the header. Just pass the list of script sections as an array reference. This allows you to specify different source files for different dialects of JavaScript. Example:

print $q->start\_html(-title=>'The Riddle of the Sphinx',

-script=>[

{ -language => 'JavaScript1.0',

-src => '/javascript/utilities10.js'

},

{ -language => 'JavaScript1.1',

-src => '/javascript/utilities11.js'

},

{ -language => 'JavaScript1.2',

-src => '/javascript/utilities12.js'

},

{ -language => 'JavaScript28.2',

-src => '/javascript/utilities219.js'

}

]

);

(If this looks a bit extreme, take my advice and stick with straight CGI scripting.)

* A **-noScript** parameter to pass some HTML that will be displayed in browsers that do not have JavaScript (or have JavaScript turned off).
* **-onLoad** and **-onUnload** parameters to register JavaScript event handlers to be executed when the page generated by your script is opened and closed respectively. Example:
* print $query->start\_html(-title=>'The Riddle of the Sphinx',
* -script=>$JSCRIPT,
* -onLoad=>'riddle\_me\_this()');

See [JavaScripting](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) for more details.

* Any additional attributes you want to incorporate into the <BODY> tag (as many as you like). This is a good way to incorporate other Netscape extensions, such as background color and wallpaper pattern. (The example above sets the page background to a vibrant blue.) You can use this feature to take advantage of new HTML features without waiting for a CGI.pm release.

**Ending an HTML Document**

print $query->end\_html

This ends an HTML document by printing the </BODY> </HTML> tags.

**Other HTML Tags**

CGI.pm provides shortcut methods for many other HTML tags. All HTML2 tags and the Netscape extensions are supported, as well as the HTML3 and HTML4 tags. Unpaired tags, paired tags, and tags that contain attributes are all supported using a simple syntax.

To see the list of HTML tags that are supported, open up the CGI.pm file and look at the functions defined in the %EXPORT\_TAGS array.

**Unpaired Tags**

Unpaired tags include <P>, <HR> and <BR>. The syntax for creating them is:

print $query->hr;

This prints out the text "<hr>".

**Paired Tags**

Paired tags include <EM>, <I> and the like. The syntax for creating them is:

print $query->em("What a silly art exhibit!");

This prints out the text "<em>What a silly art exhibit!</em>".

You can pass as many text arguments as you like: they'll be concatenated together with spaces. This allows you to create nested tags easily:

print $query->h3("The",$query->em("silly"),"art exhibit");

This creates the text:

<h3>The <em>silly</em> art exhibit</h3>

When used in conjunction with the [import](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#import) facility, the HTML shortcuts can make CGI scripts easier to read. For example:

use CGI qw/:standard/;

print h1("Road Guide"),

ol(

li(a({href=>"start.html"},"The beginning")),

li(a({href=>"middle.html"},"The middle")),

li(a({href=>"end.html"},"The end"))

);

Most HTML tags are represented as lowercase function calls. There are a few exceptions:

1. The <tr> tag used to start a new table row conflicts with the perl *translate* function tr(). Use TR() or Tr() instead.
2. The <param> tag used to pass parameters to an applet conflicts with CGI's own param() method. Use PARAM() instead.
3. The <select> tag used to create selection lists conflicts with Perl's select() function. Use Select() instead.
4. The <sub> tag used to create subscripts conflicts wit Perl's operator for creating subroutines. Use Sub() instead.

**Tags with Attributes**

To add attributes to an HTML tag, simply pass a reference to an associative array as the first argument. The keys and values of the associative array become the names and values of the attributes. For example, here's how to generate an <A> anchor link:

use CGI qw/:standard/;

print a({-href=>"bad\_art.html"},"Jump to the silly exhibit");

*<A HREF="bad\_art.html">Jump to the silly exhibit</A>*

You may dispense with the dashes in front of the attribute names if you prefer:

print img {src=>'fred.gif',align=>'LEFT'};

*<IMG ALIGN="LEFT" SRC="fred.gif">*

Sometimes an HTML tag attribute has no argument. For example, ordered lists can be marked as COMPACT, or you wish to specify that a table has a border with <TABLE BORDER>. The syntax for this is an argument that that points to an undef string:

print ol({compact=>undef},li('one'),li('two'),li('three'));

Prior to CGI.pm version 2.41, providing an empty ('') string as an attribute argument was the same as providing undef. However, this has changed in order to accomodate those who want to create tags of the form <IMG ALT="">. The difference is shown in this table:

|  |  |
| --- | --- |
| **CODE** | **RESULT** |
| img({alt=>undef}) | <IMG ALT> |
| img({alt=>''}) | <IMT ALT=""> |

**Distributive HTML Tags and Tables**

All HTML tags are distributive. If you give them an argument consisting of a **reference** to a list, the tag will be distributed across each element of the list. For example, here's one way to make an ordered list:

print ul(

li({-type=>'disc'},['Sneezy','Doc','Sleepy','Happy']);

);

This example will result in HTML output that looks like this:

<UL>

<LI TYPE="disc">Sneezy</LI>

<LI TYPE="disc">Doc</LI>

<LI TYPE="disc">Sleepy</LI>

<LI TYPE="disc">Happy</LI>

</UL>

You can take advantage of this to create HTML tables easily and naturally. Here is some code and the HTML it outputs:

use CGI qw/:standard :html3/;

print table({-border=>undef},

caption(strong('When Should You Eat Your Vegetables?')),

Tr({-align=>CENTER,-valign=>TOP},

[

th(['','Breakfast','Lunch','Dinner']),

th('Tomatoes').td(['no','yes','yes']),

th('Broccoli').td(['no','no','yes']),

th('Onions').td(['yes','yes','yes'])

]

)

);

|  |  |  |  |
| --- | --- | --- | --- |
| **When Should You Eat Your Vegetables?** | | | |
|  | **Breakfast** | **Lunch** | **Dinner** |
| **Tomatoes** | no | yes | yes |
| **Broccoli** | no | no | yes |
| **Onions** | yes | yes | yes |

If you want to produce tables programatically, you can do it this way:

use CGI qw/:standard :html3/;

@values = (1..5);

@headings = ('N','N'.sup('2'),'N'.sup('3'));

@rows = th(\@headings);

foreach $n (@values) {

push(@rows,td([$n,$n\*\*2,$n\*\*3]));

}

print table({-border=>undef,-width=>'25%'},

caption(b('Wow. I can multiply!')),

Tr(\@rows)

);

|  |  |  |
| --- | --- | --- |
| **Wow. I can multiply!** | | |
| **N** | **N2** | **N3** |
| 1 | 1 | 1 |
| 2 | 4 | 8 |
| 3 | 9 | 27 |
| 4 | 16 | 64 |
| 5 | 25 | 125 |

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Creating Forms**

*General note 1.* The various form-creating methods all return strings to the caller. These strings will contain the HTML code that will create the requested form element. You are responsible for actually printing out these strings. It's set up this way so that you can place formatting tags around the form elements.

*General note 2.* The default values that you specify for the forms are only used the **first** time the script is invoked. If there are already values present in the query string, they are used, even if blank.

If you want to change the value of a field from its previous value, you have two choices:

1. call the **param()** method to set it.
2. use the **-override** (alias **-force**) parameter. (This is a new feature in 2.15) This forces the default value to be used, regardless of the previous value of the field:
3. print $query->textfield(-name=>'favorite\_color',
4. -default=>'red',
5. -override=>1);

If you want to reset all fields to their defaults, you can:

1. Create a special *defaults* button using the **defaults()** method.
2. Create a hypertext link that calls your script without any parameters.

*General note 3.* You can put multiple forms on the same page if you wish. However, be warned that it isn't always easy to preserve state information for more than one form at a time. See [advanced techniques](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#advanced) for some hints.

*General note 4.* By popular demand, the text and labels that you provide for form elements are escaped according to HTML rules. This means that you can safely use "<CLICK ME>" as the label for a button. However, this behavior may interfere with your ability to incorporate special HTML character sequences, such as &Aacute; (Á) into your fields. If you wish to turn off automatic escaping, call the autoEscape() method with a false value immediately after creating the CGI object:

$query = new CGI;

$query->autoEscape(0);

You can turn autoescaping back on at any time with $query->autoEscape(1)

**Form Elements**

 [Opening a form](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#startform)

 [Text entry fields](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#textfield)

 [Big text entry fields](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#textarea)

 [Password fields](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#password)

 [File upload fields](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#upload)

 [Popup menus](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#menu)

 [Scrolling lists](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#scrolling_list)

 [Checkbox groups](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#checkbox_group)

 [Individual checkboxes](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#checkbox)

 [Radio button groups](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#radio)

 [Submission buttons](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#submit)

 [Reset buttons](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#reset)

 [Reset to defaults button](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#defaults)

 [Hidden fields](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#hidden)

 [Clickable Images](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#image)

 [JavaScript Buttons](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#button)

 [Autoescaping HTML](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#escape)

[Up to table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Creating An Isindex Tag**

print $query->isindex($action);

**isindex()** without any arguments returns an <ISINDEX> tag that designates your script as the URL to call. If you want the browser to call a different URL to handle the search, pass isindex() the URL you want to be called.

**Starting And Ending A Form**

print $query->startform($method,$action,$encoding);

*...various form stuff...*

print $query->endform;

**startform()** will return a <FORM> tag with the optional method, action and form encoding that you specify. **endform()** returns a </FORM> tag.

The form encoding supports the "file upload" feature of Netscape 2.0 (and higher) and Internet Explorer 4.0 (and higher). The form encoding tells the browser how to package up the contents of the form in order to transmit it across the Internet. There are two types of encoding that you can specify:

**application/x-www-form-urlencoded**

This is the type of encoding used by all browsers prior to Netscape 2.0. It is compatible with many CGI scripts and is suitable for short fields containing text data. For your convenience, CGI.pm stores the name of this encoding type in $CGI::URL\_ENCODED.

**multipart/form-data**

This is the newer type of encoding introduced by Netscape 2.0. It is suitable for forms that contain very large fields or that are intended for transferring binary data. Most importantly, it enables the "file upload" feature of Netscape 2.0 forms. For your convenience, CGI.pm stores the name of this encoding type in CGI::MULTIPART()

Forms that use this type of encoding are not easily interpreted by CGI scripts unless they use CGI.pm or another library that knows how to handle them. Unless you are using the file upload feature, there's no particular reason to use this type of encoding.

For compatability, the startform() method uses the older form of encoding by default. If you want to use the newer form of encoding By default, you can call [start\_multipart\_form()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html" \l "multipart) instead of startform().

If you plan to make use of the [JavaScript features](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting), you can provide startform() with the optional -name and/or -onSubmit parameters. -name has no effect on the display of the form, but can be used to give the form an identifier so that it can be manipulated by JavaScript functions. Provide the -onSubmit parameter in order to register some JavaScript code to be performed just before the form is submitted. This is useful for checking the validity of a form before submitting it. Your JavaScript code should return a value of "true" to let Netscape know that it can go ahead and submit the form, and "false" to abort the submission.

**Starting a Form that Uses the "File Upload" Feature**

print $query->start\_multipart\_form($method,$action,$encoding);

*...various form stuff...*

print $query->endform;

This has exactly the same usage as startform(), but it specifies form encoding type multipart/form-data as the default.

**Creating A Text Field**

*Named parameter style*

print $query->textfield(-name=>'field\_name',

-default=>'starting value',

-size=>50,

-maxlength=>80);

*Old style*

print $query->textfield('foo','starting value',50,80);

**textfield()** will return a text input field.

* The first parameter (**-name**) is the required name for the field.
* The optional second parameter (**-default**) is the starting value for the field contents.
* The optional third parameter (**-size**) is the size of the field in characters.
* The optional fourth parameter (**-maxlength**) is the maximum number of characters the field will accomodate.

As with all these methods, the field will be initialized with its previous contents from earlier invocations of the script. If you want to force in the new value, overriding the existing one, see [General note 2](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#overriding).

When the form is processed, the value of the text field can be retrieved with:

$value = $query->param('foo');

**JavaScripting:** You can also provide **-onChange, -onFocus, -onBlur, -onMouseOver, -onMouseOut** and **-onSelect** parameters to register [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) event handlers.

**Creating A Big Text Field**

*Named parameter style*

print $query->textarea(-name=>'foo',

-default=>'starting value',

-rows=>10,

-columns=>50);

*Old style*

print $query->textarea('foo','starting value',10,50);

**textarea()** is just like textfield(), but it allows you to specify rows and columns for a multiline text entry box. You can provide a starting value for the field, which can be long and contain multiple lines.

**JavaScripting:** Like textfield(), you can provide **-onChange, -onFocus, -onBlur, -onMouseOver, -onMouseOut** and **-onSelect** parameters to register [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) event handlers.

**Creating A Password Field**

*Named parameter style*

print $query->password\_field(-name=>'secret',

-value=>'starting value',

-size=>50,

-maxlength=>80);

*Old style*

print $query->password\_field('secret','starting value',50,80);

**password\_field()** is identical to textfield(), except that its contents will be starred out on the web page.

**Creating a File Upload Field**

*Named parameters style*

print $query->filefield(-name=>'uploaded\_file',

-default=>'starting value',

-size=>50,

-maxlength=>80);

*Old style*

print $query->filefield('uploaded\_file','starting value',50,80);

**filefield()** will return a form field that prompts the user to upload a file.

* The first parameter (**-name**) is the required name for the field.
* The optional second parameter (**-default**) is the starting value for the file name. This field is currently ignored by all browsers, but there's always hope!
* The optional third parameter (**-size**) is the size of the field in characters.
* The optional fourth parameter (**-maxlength**) is the maximum number of characters the field will accomodate.

filefield() will return a file upload field for use with recent browsers. The browser will prompt the remote user to select a file to transmit over the Internet to the server. Other browsers currently ignore this field.

In order to take full advantage of the file upload facility you must use the new [multipart form encoding scheme](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#multipart). You can do this either by calling [startform()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html" \l "startform) and specify an encoding type of $CGI::MULTIPART or by using the new [start\_multipart\_form()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html" \l "multipart) method. If you don't use multipart encoding, then you'll be able to retreive the name of the file selected by the remote user, but you won't be able to access its contents.

When the form is processed, you can retrieve the entered filename by calling param().

$filename = $query->param('uploaded\_file');

where "uploaded\_file" is whatever you named the file upload field. Depending on the browser version, the filename that gets returned may be the full local file path on the **remote user's** machine, or just the bare filename. If a path is provided, the follows the path conventions of the local machine.

The filename returned is also a file handle. You can read the contents of the file using standard Perl file reading calls:

# Read a text file and print it out

while (<$filename>) {

print;

}

# Copy a binary file to somewhere safe

open (OUTFILE,">>/usr/local/web/users/feedback");

while ($bytesread=read($filename,$buffer,1024)) {

print OUTFILE $buffer;

}

close $filename;

There are problems with the dual nature of the upload fields. If you use strict, then Perl will complain when you try to use a string as a filehandle. You can get around this by placing the file reading code in a block containing the no strict pragma. More seriously, it is possible for the remote user to type garbage into the upload field, in which case what you get from **param()** is not a filehandle at all, but a string.

To be safe, use the **upload()** function (new in version 2.47). When called with the name of an upload field, **upload()** returns a filehandle, or undef if the parameter is not a valid filehandle.

$fh = $query->upload('uploaded\_file');

while (<$fh>) {

print;

}

In an array context, upload() will return an array of filehandles. This makes it possible to create forms that use the same name for multiple upload fields.

This is the recommended idiom.

You can have several file upload fields in the same form, and even give them the same name if you like (in the latter case param() will return a list of file names). However, if the user attempts to upload several files with exactly the same name, CGI.pm will only return the last of them. This is a known bug.

When processing an uploaded file, CGI.pm creates a temporary file on your hard disk and passes you a file handle to that file. After you are finished with the file handle, CGI.pm unlinks (deletes) the temporary file. If you need to you can access the temporary file directly. Its name is stored inside the CGI object's "private" data, and you can access it by passing the file name to the [tmpFileName()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html" \l "tmpfilename) method:

$filename = $query->param('uploaded\_file');

$tmpfilename = $query->tmpFileName($filename);

The temporary file will be deleted automatically when your program exits unless you manually rename it. On some operating systems (such as Windows NT), you will need to close the temporary file's filehandle before your program exits. Otherwise the attempt to delete the temporary file will fail.

A potential problem with the temporary file upload feature is that the temporary file is accessible to any local user on the system. In previous versions of this module, the temporary file was world readable, meaning that anyone could peak at what was being uploaded. As of version 2.36, the modes on the temp file have been changed to read/write by owner only. Only the Web server and its CGI scripts can access the temp file. Unfortunately this means that one CGI script can spy on another! To make the temporary files **really** private, set the CGI global variable $CGI::PRIVATE\_TEMPFILES to 1. Alternatively, call the built-in function CGI::private\_tempfiles(1), or just *use CGI qw/-private\_tempfiles*. The temp file will now be unlinked as soon as it is created, making it inaccessible to other users. The **downside** of this is that you will be unable to access this temporary file directly (*tmpFileName()* will continue to return a string, but you will find no file at that location.) Further, since PRIVATE\_TEMPFILES is a global variable, its setting will affect all instances of CGI.pm if you are running mod\_perl. You can work around this limitation by declaring $CGI::PRIVATE\_TEMPFILES as a local at the top of your script.

On Windows NT, it is impossible to make a temporary file private. This is because Windows doesn't allow you to delete a file before closing it.

Usually the browser sends along some header information along with the text of the file itself. Currently the headers contain only the original file name and the MIME content type (if known). Future browsers might send other information as well (such as modification date and size). To retrieve this information, call **uploadInfo()**. It returns a reference to an associative array containing all the document headers. For example, this code fragment retrieves the MIME type of the uploaded file (be careful to use the proper capitalization for "Content-Type"!):

$filename = $query->param('uploaded\_file');

$type = $query->uploadInfo($filename)->{'Content-Type'};

unless ($type eq 'text/html') {

die "HTML FILES ONLY!";

}

**JavaScripting:** Like textfield(), filefield() accepts **-onChange, -onFocus, -onBlur, -onMouseOver, -onMouseOut** and **-onSelect** parameters to register [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) event handlers. [Caveats and potential problems in the file upload feature.](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#upload_caveats)

**Creating A Popup Menu**

*Named parameter style*

print $query->popup\_menu(-name=>'menu\_name',

-values=>[qw/eenie meenie minie/],

-labels=>{'eenie'=>'one',

'meenie'=>'two',

'minie'=>'three'},

-default=>'meenie');

print $query->popup\_menu(-name=>'menu\_name',

-values=>['eenie','meenie','minie'],

-default=>'meenie');

*Old style*

print $query->popup\_menu('menu\_name',

['eenie','meenie','minie'],'meenie',

{'eenie'=>'one','meenie'=>'two','minie'=>'three'});

**popup\_menu()** creates a menu.

* The required first argument (**-name**) is the menu's name.
* The required second argument (**-values**) is an array *reference* containing the list of menu items in the menu. You can pass the method an anonymous array, as shown in the example, or a reference to a named array, such as \@foo. If you pass a *HASH reference*, the keys will be used for the menu values, and the values will be used for the menu labels (see -labels below). However, the menu values will be in arbitrary order.
* The optional third parameter (**-default**) is the name of the default menu choice. If not specified, the first item will be the default. The value of the previous choice will be maintained across queries.
* The optional fourth parameter (**-labels**) allows you to pass a reference to an associative array containing user-visible labels for one or more of the menu items. You can use this when you want the user to see one menu string, but have the browser return your program a different one. If you don't specify this, the value string will be used instead ("eenie", "meenie" and "minie" in this example). This is equivalent to using a hash reference for the -values parameter.

When the form is processed, the selected value of the popup menu can be retrieved using:

$popup\_menu\_value = $query->param('menu\_name');

**JavaScripting:** You can provide **-onChange, -onFocus, -onMouseOver, -onMouseOut, and -onBlur** parameters to register [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) event handlers.

**Creating A Scrolling List**

*Named parameter style*

print $query->scrolling\_list(-name=>'list\_name',

-values=>['eenie','meenie','minie','moe'],

-default=>['eenie','moe'],

-size=>5,

-multiple=>'true',

-labels=>\%labels);

*Old style*

print $query->scrolling\_list('list\_name',

['eenie','meenie','minie','moe'],

['eenie','moe'],5,'true',

\%labels);

**scrolling\_list()** creates a scrolling list.

* The first and second arguments (**-name, -values**)are the list name and values, respectively. As in the popup menu, the second argument should be an array reference or hash reference. In the latter case, the values of the hash are used as the human-readable labels in the list.
* The optional third argument (**-default**)can be either a reference to a list containing the values to be selected by default, or can be a single value to select. If this argument is missing or undefined, then nothing is selected when the list first appears.
* The optional fourth argument (**-size**) is the display size of the list.
* The optional fifth argument (**-multiple**) can be set to true to allow multiple simultaneous selections.
* The option sixth argument (**-labels**) can be used to assign user-visible labels to the list items different from the ones used for the values as above. This is equivalent to passing a hash reference to -values. In this example we assume that an associative array %labels has already been created.

When this form is processed, all selected list items will be returned as a list under the parameter name 'list\_name'. The values of the selected items can be retrieved with:

@selected = $query->param('list\_name');

**JavaScripting:** You can provide **-onChange, -onFocus, -onMouseOver, -onMouseOut** and **-onBlur** parameters to register [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) event handlers.

**Creating A Group Of Related Checkboxes**

*Named parameter style*

print $query->checkbox\_group(-name=>'group\_name',

-values=>['eenie','meenie','minie','moe'],

-default=>['eenie','moe'],

-linebreak=>'true',

-labels=>\%labels);

*Old Style*

print $query->checkbox\_group('group\_name',

['eenie','meenie','minie','moe'],

['eenie','moe'],'true',\%labels);

*HTML3 Browsers Only*

print $query->checkbox\_group(-name=>'group\_name',

-values=>['eenie','meenie','minie','moe'],

-rows=>2,-columns=>2);

**checkbox\_group()** creates a list of checkboxes that are related by the same name.

* The first and second arguments (**-name, -values**) are the checkbox name and values, respectively. As in the popup menu, the second argument should be an array reference or a hash reference. These values are used for the user-readable labels printed next to the checkboxes as well as for the values passed to your script in the query string.
* The optional third argument (**-default**) can be either a reference to a list containing the values to be checked by default, or can be a single value to checked. If this argument is missing or undefined, then nothing is selected when the list first appears.
* The optional fourth argument (**-linebreak**) can be set to true to place line breaks between the checkboxes so that they appear as a vertical list. Otherwise, they will be strung together on a horizontal line. When the form is procesed, all checked boxes will be returned as a list under the parameter name 'group\_name'. The values of the "on" checkboxes can be retrieved with:
* The optional fifth argument (**-labels**) is a reference to an hash of checkbox labels. This allows you to use different strings for the user-visible button labels and the values sent to your script. In this example we assume that an associative array %labels has previously been created. This is equivalent to passing a hash reference to -values. If you don't use **-nolabels**, CGI.pm will add HTML label tag around each checkbox and its label, so a browser can identify the text as form element label properly.
* The optional parameter **-nolabels** can be used to suppress the printing of labels next to the button. This is useful if you want to capture the button elements individually and use them inside labeled HTML3 tables.
* **Browsers that understand HTML3 tables** (such as Netscape) can take advantage of the optional parameters **-rows**, and **-columns**. These parameters cause checkbox\_group() to return an HTML3 compatible table containing the checkbox group formatted with the specified number of rows and columns. You can provide just the -columns parameter if you wish; checkbox\_group will calculate the correct number of rows for you.

To include row and column headings in the returned table, you can use the **-rowheaders** and **-colheaders** parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpetation of the checkboxes -- they're still a single named unit.

When viewed with browsers that don't understand HTML3 tables, the -rows and -columns parameters will leave you with a group of buttons that may be awkwardly formatted but still useable. However, if you add row and/or column headings, the resulting text will be very hard to read.

When the form is processed, the list of checked buttons in the group can be retrieved like this:

@turned\_on = $query->param('group\_name');

This function actually returns an array of button elements. You can capture the array and do interesting things with it, such as incorporating it into your own tables or lists. The **-nolabels** option is also useful in this regard:

@h = $query->checkbox\_group(-name=>'choice',

-value=>['fee','fie','foe'],

-nolabels=>1);

create\_nice\_table(@h);

**JavaScripting:** You can provide an **-onClick** parameter to register some [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) code to be performed every time the user clicks on any of the buttons in the group.

**Creating A Standalone Checkbox**

*Named parameter list*

print $query->checkbox(-name=>'checkbox\_name',

-checked=>'checked',

-value=>'TURNED ON',

-label=>'Turn me on');

*Old style*

print $query->checkbox('checkbox\_name',1,'TURNED ON','Turn me on');

**checkbox()** is used to create an isolated checkbox that isn't logically related to any others.

* The first parameter (**-name** is the required name for the checkbox. It will also be used for the user-readable label printed next to the checkbox.
* The optional second parameter (**-checked** specifies that the checkbox is turned on by default. Aliases for this parameter are **-selected** and **-on**.
* The optional third parameter (**-value** specifies the value of the checkbox when it is checked. If not provided, the word "on" is assumed.
* The optional fourth parameter (**-label** assigns a user-visible label to the button. If not provided, the checkbox's name will be used. CGI.pm will add HTML label tag around the checkbox and its label, so a browser can identify the text as form element label properly.

The value of the checkbox can be retrieved using:

$turned\_on = $query->param('checkbox\_name');

**JavaScripting:** You can provide an -onClick parameter to register some [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) code to be performed every time the user clicks on the button.

**Creating A Radio Button Group**

*Named parameter style*

print $query->radio\_group(-name=>'group\_name',

-values=>['eenie','meenie','minie'],

-default=>'meenie',

-linebreak=>'true',

-labels=>\%labels);

*Old style*

print $query->radio\_group('group\_name',['eenie','meenie','minie'],

'meenie','true',\%labels);

*HTML3-compatible browsers only*

print $query->radio\_group(-name=>'group\_name',

-values=>['eenie','meenie','minie','moe'],

-rows=>2,-columns=>2);

**radio\_group()** creates a set of logically-related radio buttons. Turning one member of the group on turns the others off.

* The first argument (**-name** is the name of the group and is required.
* The second argument (**-values** is the list of values for the radio buttons. The values and the labels that appear on the page are identical. Pass an array *reference* in the second argument, either using an anonymous array, as shown, or by referencing a named array as in \@foo. You may also use a hash reference in order to produce human-readable labels that are different from the values that will be returned as parameters to the CGI script.
* The optional third parameter (**-default** is the value of the default button to turn on. If not specified, the first item will be the default. Specify some nonexistent value, such as "-" if you don't want any button to be turned on.
* The optional fourth parameter (**-linebreak** can be set to 'true' to put line breaks between the buttons, creating a vertical list.
* The optional fifth parameter (**-labels** specifies an associative array containing labels to be printed next to each button. If not provided the button value will be used instead. This example assumes that the associative array %labels has already been defined. This is equivalent to passing a hash reference to -values. If you don't use **-nolabels**, CGI.pm will add HTML label tag around each radio button and its label, so a browser can identify the text as form element label properly.
* The optional parameter **-nolabels** can be used to suppress the printing of labels next to the button. This is useful if you want to capture the button elements individually and use them inside labeled HTML3 tables.
* **Browsers that understand HTML3 tables** (such as Netscape) can take advantage of the optional parameters **-rows**, and **-columns**. These parameters cause radio\_group() to return an HTML3 compatible table containing the radio cluster formatted with the specified number of rows and columns. You can provide just the -columns parameter if you wish; radio\_group will calculate the correct number of rows for you.

To include row and column headings in the returned table, you can use the **-rowheader** and **-colheader** parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpetation of the radio buttons -- they're still a single named unit.

When viewed with browsers that don't understand HTML3 tables, the -rows and -columns parameters will leave you with a group of buttons that may be awkwardly formatted but still useable. However, if you add row and/or column headings, the resulting text will be very hard to read.

When the form is processed, the selected radio button can be retrieved using:

$which\_radio\_button = $query->param('group\_name');

This function actually returns an array of button elements. You can capture the array and do interesting things with it, such as incorporating it into your own tables or lists The **-nolabels** option is useful in this regard.:

@h = $query->radio\_group(-name=>'choice',

-value=>['fee','fie','foe'],

-nolabels=>1);

create\_nice\_table(@h);

**JavaScripting**: You can provide an **-onClick** parameter to register some [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) code to be performed every time the user clicks on any of the buttons in the group.

**Creating A Submit Button**

*Named parameter style*

print $query->submit(-name=>'button\_name',

-value=>'value');

*Old style*

print $query->submit('button\_name','value');

**submit()** will create the query submission button. Every form should have one of these.

* The first argument (**-name**is optional. You can give the button a name if you have several submission buttons in your form and you want to distinguish between them.
* The second argument (**-value**is also optional. This gives the button a value that will be passed to your script in the query string, and will also appear as the user-visible label.

You can figure out which of several buttons was pressed by using different values for each one:

$which\_one = $query->param('button\_name');

* You can use **-label** as an alias for **-value**. I always get confused about which of -name and -value changes the user-visible label on the button.

**JavaScripting:** You can provide an **-onClick** parameter to register some [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) code to be performed every time the user clicks on the button. You can't prevent a form from being submitted, however. You must provide an **-onSubmit** handler to the [form itself](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html) to do that.

**Creating A Reset Button**

print $query->reset

**reset()** creates the "reset" button. It undoes whatever changes the user has recently made to the form, but does **not** necessarily reset the form all the way to the defaults. See **defaults()** for that. It takes the optional label for the button ("Reset" by default). **JavaScripting:** You can provide an **-onClick** parameter to register some [JavaScript](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) code to be performed every time the user clicks on the button.

**Creating A Defaults Button**

print $query->defaults('button\_label')

**defaults()** creates "reset to defaults" button. It takes the optional label for the button ("Defaults" by default). When the user presses this button, the form will automagically be cleared entirely and set to the defaults you specify in your script, just as it was the first time it was called.

**Creating A Hidden Field**

*Named parameter style*

print $query->hidden(-name=>'hidden\_name',

-default=>['value1','value2'...]);

*Old style*

print $query->hidden('hidden\_name','value1','value2'...);

**hidden()** produces a text field that can't be seen by the user. It is useful for passing state variable information from one invocation of the script to the next.

* The first argument (**-name**) is required and specifies the name of this field.
* The second and subsequent arguments specify the value for the hidden field. This is a quick and dirty way of passing perl arrays through forms. If you use the named parameter style, you must provide the parameter **-default** and an array reference here.

**As of version 2.0 I have changed the behavior of hidden fields once again. Read this if you use hidden fields.**

Hidden fields used to behave differently from all other fields: the provided default values always overrode the "sticky" values. This was the behavior people seemed to expect, however it turns out to make it harder to write state-maintaining forms such as shopping cart programs. Therefore I have made the behavior consistent with other fields.

Just like all the other form elements, the value of a hidden field is "sticky". If you want to replace a hidden field with some other values after the script has been called once you'll have to do it manually before writing out the form element:

$query->param('hidden\_name','new','values','here');

print $query->hidden('hidden\_name');

Fetch the value of a hidden field this way:

$hidden\_value = $query->param('hidden\_name');

-or (for values created with arrays)-

@hidden\_values = $query->param('hidden\_name');

**Creating a Clickable Image Button**

*Named parameter style*

print $query->image\_button(-name=>'button\_name',

-src=>'/images/NYNY.gif',

-align=>'MIDDLE');

*Old style*

print $query->image\_button('button\_name','/source/URL','MIDDLE');

**image\_button()** produces an inline image that acts as a submission button. When selected, the form is submitted and the clicked (x,y) coordinates are submitted as well.

* The first argument(**-name** is required and specifies the name of this field.
* The second argument (**-src**specifies the URL of the image to display. It must be one of the types supported by inline images (e.g. GIF), but can be any local or remote URL.
* The third argument (**-align**is anything you might want to use in the ALIGN attribute, such as TOP, BOTTOM, LEFT, RIGHT or MIDDLE. This field is optional.

When the image is clicked, the results are passed to your script in two parameters named "button\_name.x" and "button\_name.y", where "button\_name" is the name of the image button.

$x = $query->param('button\_name.x');

$y = $query->param('button\_name.y');

**JavaScripting:** Current versions of JavaScript do not honor the -onClick handler, unlike other buttons.

**Creating a JavaScript Button**

*Named parameter style*

print $query->button(-name=>'button1',

-value=>'Click Me',

-onClick=>'doButton(this)');

*Old style*

print $query->image\_button('button1','Click Me','doButton(this)');

**button()** creates a JavaScript button. When the button is pressed, the JavaScript code pointed to by the -onClick parameter is executed. This only works with Netscape 2.0 and higher. Other browsers do not recognize JavaScript and probably won't even display the button.

* The first argument(**-name** is required and specifies the name of this field.
* The second argument (**-value** gives the button a value, and will be used as the user-visible label on the button.
* The third argument (**-onClick** is any valid JavaScript code. It's usually a call to a JavaScript function defined somewhere else (see the [start\_html()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html" \l "html) method), but can be any JavaScript you like. Multiple lines are allowed, but you must be careful not to include any double quotes in the JavaScript text.

See [JavaScripting](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#javascripting) for more information.

**Controlling HTML Autoescaping**

By default, if you use a special HTML character such as >, < or & as the label or value of a button, it will be escaped using the appropriate HTML escape sequence (e.g. &gt;). This lets you use anything at all for the text of a form field without worrying about breaking the HTML document. However, it may also interfere with your ability to use special characters, such as Á as default contents of fields. You can turn this feature on and off with the method autoEscape().

Use

$query->autoEscape(0);

to turn automatic HTML escaping off, and

$query->autoEscape(1);

to turn it back on.

**Importing CGI Methods**

A large number of scripts allocate only a single query object, use it to read parameters or to create a fill-out form, and then discard it. For this type of script, it may be handy to import CGI module methods into your name space. The most common syntax for this is:

use CGI qw(:standard);

This imports the standard methods into your namespace. Now instead of getting parameters like this:

use CGI;

$dinner = $query->param('entree');

You can do it like this:

use CGI qw(:standard);

$dinner = param('entree');

Similarly, instead of creating a form like this:

print $query->start\_form,

"Check here if you're happy: ",

$query->checkbox(-name=>'happy',-value=>'Y',-checked=>1),

"<P>",

$query->submit,

$query->end\_form;

You can create it like this:

print start\_form,

"Check here if you're happy: ",

checkbox(-name=>'happy',-value=>'Y',-checked=>1),

p,

submit,

end\_form;

Even though there's no CGI object in view in the second example, state is maintained using an implicit CGI object that's created automatically. The form elements created this way are sticky, just as before. If you need to get at the implicit CGI object directly, you can refer to it as:

$CGI::Q;

The **use CGI** statement is used to import method names into the current name space. There is a slight overhead for each name you import, but ordinarily is nothing to worry about. You can import selected method names like this:

use CGI qw(header start\_html end\_html);

Ordinarily, however, you'll want to import groups of methods using export tags. Export tags refer to sets of logically related methods which are imported as a group with **use**. Tags are distinguished from ordinary methods by beginning with a ":" character. This example imports the methods dealing with the CGI protocol (param() and the like) as well as shortcuts that generate HTML2-compliant tags:

use CGI qw(:cgi :html2);

Currently there are 8 method families defined in CGI.pm. They are:

*:cgi*

These are all the tags that support one feature or another of the CGI protocol, including param(), path\_info(), cookie(), request\_method(), header() and the like.

*:form*

These are all the form element-generating methods, including start\_form(), textfield(), etc.

*:html2*

These are HTML2-defined shortcuts such as br(), p() and head(). It also includes such things as start\_html() and end\_html() that aren't exactly HTML2, but are close enough.

*:html3*

These contain various HTML3 tags for tables, frames, super- and subscripts, applets and other objects.

*:html4*

These contain various HTML4 tags, including table headers and footers.

*:netscape*

These are Netscape extensions not included in the HTML3 category including blink() and center().

*:html*

These are all the HTML generating shortcuts, comprising the union of *html2, html3,* and *netscape*.

*:multipart*

These are various functions that simplify creating documents of the various multipart MIME types, and are useful for implementing server push.

*:standard*

This is the union of *html2, html3, html4, form,* and *:cgi*.

*:all*

This imports all the public methods into your namespace!

**Pragmas**

In addition to importing individual methods and method families, *use CGI* recognizes several pragmas, all proceeded by dashes.

**-any**

When you *use CGI -any*, then any method that the query object doesn't recognize will be interpreted as a new HTML tag. This allows you to support the next *ad hoc* Netscape or Microsoft HTML extension. For example, to support Netscape's latest tag, <GRADIENT> (which causes the user's desktop to be flooded with a rotating gradient fill until his machine reboots), you can use something like this:

use CGI qw(-any);

$q=new CGI;

print $q->gradient({speed=>'fast',start=>'red',end=>'blue'});

Since using *any* causes any mistyped method name to be interpreted as an HTML tag, use it with care or not at all.

**-compile**

This causes the indicated autoloaded methods to be compiled up front, rather than deferred to later. This is useful for scripts that run for an extended period of time under FastCGI or mod\_perl, and for those destined to be crunched by Malcom Beattie's Perl compiler. Use it in conjunction with the methods or method familes you plan to use.

use CGI qw(-compile :standard :html3);

or even

use CGI qw(-compile :all);

Note that using the -compile pragma in this way will always have the effect of importing the compiled functions into the current namespace. If you want to compile without importing use the [compile()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#compile) method instead.

**-autoload**

Overrides the autoloader so that any function in your program that is not recognized is referred to CGI.pm for possible evaluation. This allows you to use all the CGI.pm functions without adding them to your symbol table, which is of concern for mod\_perl users who are worried about memory consumption. **Warning:** when *-autoload* is in effect, you cannot use "poetry mode" (functions without the parenthesis). Use *hr()* rather than *hr*, or add something like *use subs qw/hr p header/* to the top of your script.

**-nosticky**

Turns off "sticky" behavior in fill-out forms. Every form element will act as if you passed -override.

**-no\_xhtml**

By default, CGI.pm versions 2.69 and higher emit XHTML (<http://www.w3.org/TR/xhtml1/>). The -no\_xhtml pragma disables this feature. Thanks to Michalis Kabrianis <kabrianis@hellug.gr> for this feature.

**-nph**

This makes CGI.pm produce a header appropriate for an NPH (no parsed header) script. You may need to do other things as well to tell the server that the script is NPH. See the [discussion of NPH scripts](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#nph) below.

**-oldstyle\_urls**

Separate the name=value pairs in CGI parameter query strings emitted by self\_url() and query\_string() with ampersands. Otherwise, CGI.pm emits HTML-compliant semicolons. If you use this form, be sure to escape ampersands into HTML entities with escapeHTML. Example:

$href = $q->self\_url();

$href = escapeHTML($href);

print [I'm talking to myself](http://pages.cs.wisc.edu/~beechung/ref/$href)

**-newstyle\_urls**

Separate the name=value pairs in CGI parameter query strings with semicolons rather than ampersands. For example:

name=fred;age=24;favorite\_color=3

As of version 2.64, this is the default style.

**-no\_debug**

This turns off the command-line processing features. If you want to run a CGI.pm script from the command line to produce HTML, and you don't want it interpreting arguments on the command line as CGI name=value arguments, then use this pragma:

use CGI qw(-no\_debug :standard);

**-debug**

This turns on full debugging. In addition to reading CGI arguments from the command-line processing, CGI.pm will pause and try to read arguments from STDIN, producing the message "(offline mode: enter name=value pairs on standard input)" features.

See [debugging](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#debugging) for more details.

**-private\_tempfiles**

CGI.pm can process uploaded file. Ordinarily it spools the uploaded file to a temporary directory, then deletes the file when done. However, this opens the risk of eavesdropping as described in the [file upload section.](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#upload) Another CGI script author could peek at this data during the upload, even if it is confidential information. On Unix systems, the **-private\_tempfiles** pragma will cause the temporary file to be unlinked as soon as it is opened and before any data is written into it, eliminating the risk of eavesdropping.

**Special Forms for Importing HTML-Tag Functions**

Many of the methods generate HTML tags. As described below, tag functions automatically generate both the opening and closing tags. For example:

print h1('Level 1 Header');

produces

<H1>Level 1 Header</H1>

There will be some times when you want to produce the start and end tags yourself. In this case, you can use the form start\_I*tag\_name* and end\_I*tag\_name*, as in:

print start\_h1,'Level 1 Header',end\_h1;

With a few exceptions (described below), start\_*tag\_name* and end\_I*tag\_name* functions are not generated automatically when you *use CGI*. However, you can specify the tags you want to generate *start/end* functions for by putting an asterisk in front of their name, or, alternatively, requesting either "start\_*tag\_name*" or "end\_*tag\_name*" in the import list.

Example:

use CGI qw/:standard \*table start\_ul/;

In this example, the following functions are generated in addition to the standard ones:

1. start\_table() (generates a <TABLE> tag)
2. end\_table() (generates a </TABLE> tag)
3. start\_ul() (generates a <UL> tag)
4. end\_ul() (generates a </UL> tag)

**AUTOESCAPING HTML**

By default, all HTML that are emitted by the form-generating functions are passed through a function called escapeHTML():

$escaped\_string = escapeHTML("unescaped string");

Provided that you have specified a character set of ISO-8859-1 (the default), the standard HTML escaping rules will be used. The "<" character becomes "&lt;", ">" becomes "&gt;", "&" becomes "&amp;", and the quote character becomes "&quot;". In addition, the hexadecimal 0x8b and 0x9b characters, which many windows-based browsers interpret as the left and right angle-bracket characters, are replaced by their numeric HTML entities ("&#139" and "&#155;"). If you manually change the charset, either by calling the charset() method explicitly or by passing a -charset argument to header(), then **all** characters will be replaced by their numeric entities, since CGI.pm has no lookup table for all the possible encodings.

Autoescaping does not apply to other HTML-generating functions, such as h1(). You should call escapeHTML() yourself on any data that is passed in from the outside, such as nasty text that people may enter into guestbooks.

To change the character set, use charset(). To turn autoescaping off completely, use autoescape():

$charset = charset([$charset]); # Get or set the current character set.

$flag = autoEscape([$flag]); # Get or set the value of the autoescape flag.

**PRETTY-PRINTING HTML**

By default, all the HTML produced by these functions comes out as one long line without carriage returns or indentation. This is yuck, but it does reduce the size of the documents by 10-20%. To get pretty-printed output, please use *CGI::Pretty*, a subclass contributed by [Brian Paulsen](mailto:bpaulsen@lehman.com).

**Optional Utility Functions**

In addition to the standard imported functions, there are a few optional functions that you must request by name if you want them. They were originally intended for internal use only, but are now made available by popular request.

**escape(), unescape()**

use CGI qw/escape unescape/;

$q = escape('This $string contains ~wonderful~ characters');

$u = unescape($q);

These functions escape and unescape strings according to the URL hex escape rules. For example, the space character will be converted into the string "%20".

**escapeHTML(), unescapeHTML()**

use CGI qw/escapeHTML unescapeHTML/;

$q = escapeHTML('This string is <illegal> html!');

$u = unescapeHTML($q);

These functions escape and unescape strings according to the HTML character entity rules. For example, the character < will be escaped as &lt;.

**compile()**

Ordinarily CGI.pm autoloads most of its functions on an as-needed basis. This speeds up the loading time by deferring the compilation phase. However, if you are using mod\_perl, FastCGI or another system that uses a persistent Perl interpreter, you will want to precompile the methods at initialization time. To accomplish this, call the package function **compile()** like this:

use CGI ();

CGI->compile(':all');

The arguments to **compile()** are a list of method names or sets, and are identical to those accepted by the use operator.

**Debugging**

If you are running the script from the command line or in the perl debugger, you can pass the script a list of keywords or parameter=value pairs on the command line or from standard input (you don't have to worry about tricking your script into reading from environment variables). You can pass keywords like this:

my\_script.pl keyword1 keyword2 keyword3

*or this:*

my\_script.pl keyword1+keyword2+keyword3

*or this:*

my\_script.pl name1=value1 name2=value2

*or this:*

my\_script.pl name1=value1&name2=value2

If you pass the **-debug** pragma to CGI.pm, you can send CGI name-value pairs as newline-delimited parameters on standard input:

% my\_script.pl

first\_name=fred

last\_name=flintstone

occupation='granite miner'

^D

When debugging, you can use quotation marks and the backslash character to escape spaces and other funny characters in exactly the way you would in the shell (which isn't surprising since CGI.pm uses "shellwords.pl" internally). This lets you do this sort of thing:

my\_script.pl 'name 1=I am a long value' name\ 2=two\ words

If you run a script that uses CGI.pm from the command line and fail to provide it with any arguments, it will print out the line

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

then appear to hang. In fact, the library is waiting for you to give it some parameters to process on its standard input. If you want to give it some parameters, enter them as shown above, then indicate that you're finished with input by pressing ^D (^Z on NT/DOS systems). If you don't want to give CGI.pm parameters, just press ^D.

You can suppress this behavior in any of the following ways:

1. Call the script with an empty parameter.

Example:

my\_script.pl ''

2. Redirect standard input from /dev/null or an empty file.

Example:

my\_script.pl </dev/null

3. Include "-no\_debug" in the list of symbols to import on the "use" line.

Example:

use CGI qw/:standard -no\_debug/;

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Dumping Out All The Name/Value Pairs**

The **Dump()** method produces a string consisting of all the query's name/value pairs formatted nicely as a nested list. This is useful for debugging purposes:

print $query->Dump

Produces something that looks like this:

<UL>

<LI>name1

<UL>

<LI>value1

<LI>value2

</UL>

<LI>name2

<UL>

<LI>value1

</UL>

</UL>

You can achieve the same effect by incorporating the CGI object directly into a string, as in:

print "<H2>Current Contents:</H2>\n$query\n";

**HTTP Session Variables**

Some of the more useful environment variables can be fetched through this interface. The methods are as follows:

Accept()

Return a list of MIME types that the remote browser accepts. If you give this method a single argument corresponding to a MIME type, as in $query->Accept('text/html'), it will return a floating point value corresponding to the browser's preference for this type from 0.0 (don't want) to 1.0. Glob types (e.g. text/\*) in the browser's accept list are handled correctly. Note the capitalization of the initial letter. This avoids conflict with the Perl built-in accept().

auth\_type()

Return the authorization type, if protection is active. Example "Basic".

raw\_cookie()

Returns the "magic cookie" maintained by Netscape 1.1 and higher in a raw state. You'll probably want to use [cookie()](http://pages.cs.wisc.edu/~beechung/ref/cookies) instead, which gives you a high-level interface to the cookie functions. Called with no parameters, raw\_cookie() returns the entire cookie structure, which may consist of several cookies appended together (you can recover individual cookies by splitting on the "; " sequence. Called with the name of a cookie, returns the unescaped value of the cookie as set by the server. This may be useful for retrieving cookies that your script did not set.

path\_info()

Returns additional path information from the script URL. E.G. fetching /cgi-bin/your\_script/additional/stuff will result in $query->path\_info() returning "/additional/stuff". In addition to reading the path information, you can set it by giving path\_info() an optional string argument. The argument is expected to begin with a "/". If not present, one will be added for you. The new path information will be returned by subsequent calls to path\_info(), and will be incorporated into the URL generated by self\_url().

path\_translated()

As per path\_info() but returns the additional path information translated into a physical path, e.g. "/usr/local/etc/httpd/htdocs/additional/stuff". You cannot change the path\_translated, nor will setting the additional path information change this value. The reason for this restriction is that the translation of path information into a physical path is ordinarily done by the server in a layer that is inaccessible to CGI scripts.

query\_string()

Returns a query string suitable for maintaining state.

referer()

Return the URL of the page the browser was viewing prior to fetching your script. Not available for all browsers.

remote\_addr()

Return the dotted IP address of the remote host.

remote\_ident()

Return the identity-checking information from the remote host. Only available if the remote host has the identd daemon turned on.

remote\_host()

Returns either the remote host name or IP address. if the former is unavailable.

remote\_user()

Return the name given by the remote user during password authorization.

request\_method()

Return the HTTP method used to request your script's URL, usually one of GET, POST, or HEAD.

script\_name()

Return the script name as a partial URL, for self-refering scripts.

server\_name()

Return the name of the WWW server the script is running under.

server\_software()

Return the name and version of the server software.

virtual\_host()

When using the virtual host feature of some servers, returns the name of the virtual host the browser is accessing.

server\_port()

Return the communications port the server is using.

user\_agent()

Returns the identity of the remote user's browser software, e.g. "Mozilla/1.1N (Macintosh; I; 68K)"

user\_name()

Attempts to obtain the remote user's name, using a variety of environment variables. This only works with older browsers such as Mosaic. Netscape does not reliably report the user name!

http()

Called with no arguments returns the list of HTTP environment variables, including such things as HTTP\_USER\_AGENT, HTTP\_ACCEPT\_LANGUAGE, and HTTP\_ACCEPT\_CHARSET, corresponding to the like-named HTTP header fields in the request. Called with the name of an HTTP header field, returns its value. Capitalization and the use of hyphens versus underscores are not significant.

For example, all three of these examples are equivalent:

$requested\_language = $q->http('Accept-language');

$requested\_language = $q->http('Accept\_language');

$requested\_language = $q->http('HTTP\_ACCEPT\_LANGUAGE');

https()

The same as http(), but operates on the HTTPS environment variables present when the SSL protocol is in effect. Can be used to determine whether SSL is turned on.

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**HTTP Cookies**

Netscape browsers versions 1.1 and higher, and all versions of Internet Explorer support a so-called "cookie" designed to help maintain state within a browser session. CGI.pm has several methods that support cookies.

A cookie is a name=value pair much like the named parameters in a CGI query string. CGI scripts create one or more cookies and send them to the browser in the HTTP header. The browser maintains a list of cookies that belong to a particular Web server, and returns them to the CGI script during subsequent interactions.

In addition to the required name=value pair, each cookie has several optional attributes:

an expiration time

This is a time/date string (in a special GMT format) that indicates when a cookie expires. The cookie will be saved and returned to your script until this expiration date is reached if the user exits the browser and restarts it. If an expiration date isn't specified, the cookie will remain active until the user quits the browser.

Negative expiration times (e.g. "-1d") cause some browsers to delete the cookie from its persistent store. This is a poorly documented feature.

a domain

This is a partial or complete domain name for which the cookie is valid. The browser will return the cookie to any host that matches the partial domain name. For example, if you specify a domain name of ".capricorn.com", then the browser will return the cookie to Web servers running on any of the machines "www.capricorn.com", "www2.capricorn.com", "feckless.capricorn.com", etc. Domain names must contain at least two periods to prevent attempts to match on top level domains like ".edu". If no domain is specified, then the browser will only return the cookie to servers on the host the cookie originated from.

a path

If you provide a cookie path attribute, the browser will check it against your script's URL before returning the cookie. For example, if you specify the path "/cgi-bin", then the cookie will be returned to each of the scripts "/cgi-bin/tally.pl", "/cgi-bin/order.pl", and "/cgi-bin/customer\_service/complain.pl", but not to the script "/cgi-private/site\_admin.pl". By default, path is set to "/", which causes the cookie to be sent to any CGI script on your site.

a "secure" flag

If the "secure" attribute is set, the cookie will only be sent to your script if the CGI request is occurring on a secure channel, such as SSL.

The interface to HTTP cookies is the **cookie()** method:

$cookie = $query->cookie(-name=>'sessionID',

-value=>'xyzzy',

-expires=>'+1h',

-path=>'/cgi-bin/database',

-domain=>'.capricorn.org',

-secure=>1);

print $query->header(-cookie=>$cookie);

**cookie()** creates a new cookie. Its parameters include:

**-name**

The name of the cookie (required). This can be any string at all. Although Netscape limits its cookie names to non-whitespace alphanumeric characters, CGI.pm removes this restriction by escaping and unescaping cookies behind the scenes.

**-value**

The value of the cookie. This can be any scalar value, array reference, or even associative array reference. For example, you can store an entire associative array into a cookie this way:

$cookie=$query->cookie(-name=>'family information',

-value=>\%childrens\_ages);

**-path**

The optional partial path for which this cookie will be valid, as described above.

**-domain**

The optional partial domain for which this cookie will be valid, as described above.

**-expires**

The optional expiration date for this cookie. The format is as described in the section on the **header()** method:

"+1h" one hour from now

**-secure**

If set to true, this cookie will only be used within a secure SSL session.

The cookie created by **cookie()** must be incorporated into the HTTP header within the string returned by the [header()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#header) method:

print $query->header(-cookie=>$my\_cookie);

To create multiple cookies, give header() an array reference:

$cookie1 = $query->cookie(-name=>'riddle\_name',

-value=>"The Sphynx's Question");

$cookie2 = $query->cookie(-name=>'answers',

-value=>\%answers);

print $query->header(-cookie=>[$cookie1,$cookie2]);

To retrieve a cookie, request it by name by calling cookie() method without the **-value** parameter:

use CGI;

$query = new CGI;

%answers = $query->cookie('answers');

# $query->cookie(-name=>'answers') works too!

To retrieve the names of all cookies passed to your script, call **cookie()** without any parameters. This allows you to iterate through all cookies:

foreach $name ($query->cookie()) {

print $query->cookie($name);

}

The cookie and CGI namespaces are separate. If you have a parameter named 'answers' and a cookie named 'answers', the values retrieved by param() and cookie() are independent of each other. However, it's simple to turn a CGI parameter into a cookie, and vice-versa:

# turn a CGI parameter into a cookie

$c=$q->cookie(-name=>'answers',-value=>[$q->param('answers')]);

# vice-versa

$q->param(-name=>'answers',-value=>[$q->cookie('answers')]);

See the [cookie.cgi](http://pages.cs.wisc.edu/~beechung/ref/examples/cookie.cgi) example script for some ideas on how to use cookies effectively.

**NOTE:** There are some limitations on cookies. Here is what RFC2109, section 6.3, states:

Practical user agent implementations have limits on the number and

size of cookies that they can store. In general, user agents' cookie

support should have no fixed limits. They should strive to store as

many frequently-used cookies as possible. Furthermore, general-use

user agents should provide each of the following minimum capabilities

individually, although not necessarily simultaneously:

\* at least 300 cookies

\* at least 4096 bytes per cookie (as measured by the size of the

characters that comprise the cookie non-terminal in the syntax

description of the Set-Cookie header)

\* at least 20 cookies per unique host or domain name

User agents created for specific purposes or for limited-capacity

devices should provide at least 20 cookies of 4096 bytes, to ensure

that the user can interact with a session-based origin server.

The information in a Set-Cookie response header must be retained in

its entirety. If for some reason there is inadequate space to store

the cookie, it must be discarded, not truncated.

Applications should use as few and as small cookies as possible, and

they should cope gracefully with the loss of a cookie.

Unfortunately, some browsers appear to have limits that are more restrictive than those given in the RFC. If you need to store a lot of information, it's probably better to create a unique session ID, store it in a cookie, and use the session ID to locate an external file/database saved on the server's side of the connection.

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Support for Frames**

CGI.pm contains support for [HTML frames](http://home.netscape.com/assist/net_sites/frames.html), a feature of Netscape 2.0 and higher, and Internet Explorer 3.0 and higher. Frames are supported in two ways:

1. You can provide the name of a new or preexisting frame in the startform() and start\_multipart\_form() methods using the -target parameter. When the form is submitted, the output will be redirected to the indicated frame:
2. print $query->start\_form(-target=>'result\_frame');

1. You can direct the output of a script into a new window or into a preexisting named frame by providing the name of the frame as a -target argument in the header method. For example, the following code will pop up a new window and display the script's output:
2. $query = new CGI;
3. print $query->header(-target=>'\_blank');

This feature is a non-standard extension to HTTP which is supported by Netscape browsers, but **not by Internet Explorer**.

Using frames effectively can be tricky. To create a proper frameset in which the query and response are displayed side-by-side requires you to divide the script into three functional sections. The first section should create the <frameset> declaration and exit. The second section is responsible for creating the query form and directing it into the one frame. The third section is responsible for creating the response and directing it into a different frame.

[The examples directory](http://pages.cs.wisc.edu/~beechung/ref/examples/) contains a script called [popup.cgi](http://pages.cs.wisc.edu/~beechung/ref/examples/popup.cgi) that demonstrates a simple popup window. [frameset.cgi](http://pages.cs.wisc.edu/~beechung/ref/examples/frameset.cgi) provides a skeleton script for creating side-by-side query/result frame sets.

**Support for JavaScript**

Netscape versions 2.0 and higher incorporate an interpreted language called JavaScript. Internet Explorer, 3.0 and higher, supports a closely-related dialect called JScript. JavaScript isn't the same as Java, and certainly isn't at all the same as Perl, which is a great pity. JavaScript allows you to programatically change the contents of fill-out forms, create new windows, and pop up dialog box from within Netscape itself. From the point of view of CGI scripting, JavaScript is quite useful for validating fill-out forms prior to submitting them.

You'll need to know JavaScript in order to use it. The [Netscape JavaScript manual](http://home.netscape.com/eng/mozilla/2.0/handbook/javascript/) contains a good tutorial and reference guide to the JavaScript programming language.

The usual way to use JavaScript is to define a set of functions in a <SCRIPT> block inside the HTML header and then to register event handlers in the various elements of the page. Events include such things as the mouse passing over a form element, a button being clicked, the contents of a text field changing, or a form being submitted. When an event occurs that involves an element that has registered an event handler, its associated JavaScript code gets called.

The elements that can register event handlers include the <BODY> of an HTML document, hypertext links, all the various elements of a fill-out form, and the form itself. There are a large number of events, and each applies only to the elements for which it is relevant. Here is a partial list:

**onLoad**

The browser is loading the current document. Valid in:

* The HTML <BODY> section only.

**onUnload**

The browser is closing the current page or frame. Valid for:

* The HTML <BODY> section only.

**onSubmit**

The user has pressed the submit button of a form. This event happens just before the form is submitted, and your function can return a value of *false* in order to abort the submission. Valid for:

* Forms only.

**onClick**

The mouse has clicked on an item in a fill-out form. Valid for:

* Buttons (including submit, reset, and image buttons)
* Checkboxes
* Radio buttons

**onChange**

The user has changed the contents of a field. Valid for:

* Text fields
* Text areas
* Password fields
* File fields
* Popup Menus
* Scrolling lists

**onFocus**

The user has selected a field to work with. Valid for:

* Text fields
* Text areas
* Password fields
* File fields
* Popup Menus
* Scrolling lists

**onBlur**

The user has deselected a field (gone to work somewhere else). Valid for:

* Text fields
* Text areas
* Password fields
* File fields
* Popup Menus
* Scrolling lists

**onSelect**

The user has changed the part of a text field that is selected. Valid for:

* Text fields
* Text areas
* Password fields
* File fields

**onMouseOver**

The mouse has moved over an element.

* Text fields
* Text areas
* Password fields
* File fields
* Popup Menus
* Scrolling lists

**onMouseOut**

The mouse has moved off an element.

* Text fields
* Text areas
* Password fields
* File fields
* Popup Menus
* Scrolling lists

In order to register a JavaScript event handler with an HTML element, just use the event name as a parameter when you call the corresponding CGI method. For example, to have your validateAge() JavaScript code executed every time the textfield named "age" changes, generate the field like this:

print $q->textfield(-name=>'age',-onChange=>"validateAge(this)");

This example assumes that you've already declared the validateAge() function by incorporating it into a <SCRIPT> block. The CGI.pm [start\_html()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html" \l "html) method provides a convenient way to create this section.

Similarly, you can create a form that checks itself over for consistency and alerts the user if some essential value is missing by creating it this way:

print $q->startform(-onSubmit=>"validateMe(this)");

See the [javascript.cgi](http://pages.cs.wisc.edu/~beechung/ref/examples/javascript.cgi) script for a demonstration of how this all works.

The JavaScript "standard" is still evolving, which means that new handlers may be added in the future, or may be present in some browsers and not in others. You do not need to wait for a new version of CGI.pm to use new event handlers. Just like any other tag attribute they will produce syntactically correct HTML. For instance, if Microsoft invents a new event handler called **onInterplanetaryDisaster**, you can install a handler for it with:

print button(-name=>'bail out',-onInterPlaneteryDisaster=>"alert('uh oh')");

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Limited Support for Cascading Style Sheets**

CGI.pm has limited support for HTML3's cascading style sheets (css). To incorporate a stylesheet into your document, pass the **start\_html()** method a **-style** parameter. The value of this parameter may be a scalar, in which case it is incorporated directly into a <STYLE> section, or it may be a hash reference. In the latter case you should provide the hash with one or more of **-src** or **-code**. **-src** points to a URL where an externally-defined stylesheet can be found. **-code** points to a scalar value to be incorporated into a <STYLE> section. Style definitions in **-code** override similarly-named ones in **-src**, hence the name "cascading."

You may also specify the MIME type of the stylesheet by including an optional **-type** parameter in the hash pointed to by **-style**. If not specified, the type defaults to 'text/css'.

To refer to a style within the body of your document, add the **-class** parameter to any HTML element:

print h1({-class=>'Fancy'},'Welcome to the Party');

Or define styles on the fly with the **-style** parameter:

print h1({-style=>'Color: red;'},'Welcome to Hell');

You may also use the new **span()** element to apply a style to a section of text:

print span({-style=>'Color: red;'},

h1('Welcome to Hell'),

"Where did that handbasket get to?"

);

Note that you must import the ":html3" definitions to get the **span()** and **style()** methods.

You won't be able to do much with this unless you understand the CSS specification. A more intuitive subclassable library for cascading style sheets in Perl is in the works, but until then, please read the CSS specification at <http://www.w3.org/pub/WWW/Style/> to find out how to use these features. Here's a final example to get you started.

use CGI qw/:standard :html3/;

#here's a stylesheet incorporated directly into the page

$newStyle=<<END;

<!--

P.Tip {

margin-right: 50pt;

margin-left: 50pt;

color: red;

}

P.Alert {

font-size: 30pt;

font-family: sans-serif;

color: red;

}

-->

END

print header();

print start\_html( -title=>'CGI with Style',

-style=>{-src=>'http://www.capricorn.com/style/st1.css',

-code=>$newStyle}

);

print h1('CGI with Style'),

p({-class=>'Tip'},

"Better read the cascading style sheet spec before playing with this!"

),

span({-style=>'color: magenta'},"Look Mom, no hands!",

p(),

"Whooo wee!"

);

print end\_html;

Pass an array reference to **-style** in order to incorporate multiple stylesheets into your document.

Should you wish to incorporate a verbatim stylesheet that includes arbitrary formatting in the header, you may pass a -verbatim tag to the -style hash, as follows:

print $q->start\_html (-STYLE => {-verbatim => '@import

url("/server-common/css/'.$cssFile.'");',

-src => '/server-common/css/core.css'});

This will generate HTML like this:

<link rel="stylesheet" type="text/css"

href="/server-common/css/core.css">

<style type="text/css">

@import url("/server-common/css/main.css");

</style>

Any additional arguments passed in the -style value will be incorporated into the <link> tag. For example:

start\_html(-style=>{-src=>['/styles/print.css','/styles/layout.css'],

-media => 'all'});

This will give:

<link rel="stylesheet" type="text/css" href="/styles/print.css" media="all"/>

<link rel="stylesheet" type="text/css" href="/styles/layout.css" media="all"/>

To make more complicated <link> tags, use the Link() function and pass it to start\_html() in the -head argument, as in:

@h = (Link({-rel=>'stylesheet',-type=>'text/css',-src=>'/ss/ss.css',-media=>'all'}),

Link({-rel=>'stylesheet',-type=>'text/css',-src=>'/ss/fred.css',-media=>'paper'}));

print start\_html({-head=>\@h})

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Using NPH Scripts**

NPH, or "no-parsed-header", scripts bypass the server completely by sending the complete HTTP header directly to the browser. This has slight performance benefits, but is of most use for taking advantage of HTTP extensions that are not directly supported by your server, such as server push and PICS headers.

Servers use a variety of conventions for designating CGI scripts as NPH. IIS and many Unix servers look at the beginning of the script's name for the prefix "nph-".

CGI.pm supports NPH scripts with a special NPH mode. When in this mode, CGI.pm will output the necessary extra header information when the header() and redirect() methods are called.

**Important:** If you use the Microsoft Internet Information Server, you *must* designate your script as an NPH script. Otherwise many of CGI.pm's features, such as redirection and the ability to output non-HTML files, will fail. However, after applying Service Pack 6, NPH scripts *do not work at all* on IIS without a special patch from Microsoft. See [Knowledgebase article Q280/3/31 Non-Parsed Headers Stripped From CGI Applications That Have nph- Prefix in Name](http://support.microsoft.com/support/kb/articles/Q280/3/41.ASP)

There are a number of ways to put CGI.pm into NPH mode:

In the **use** statement:

Simply add "-nph" to the list of symbols to be imported into your script:

use CGI qw(:standard -nph)

By calling the **nph()** method:

Call **nph()** with a non-zero parameter at any point after using CGI.pm in your program.

CGI->nph(1)

By using **-nph** parameters in the **header()** and **redirect()** statements:

print $q->header(-nph=>1);

**Advanced Techniques**

**A Script that Saves Some Information to a File and Restores It**

This script will save its state to a file of the user's choosing when the "save" button is pressed, and will restore its state when the "restore" button is pressed. Notice that *it's very important to check the file name* for shell metacharacters so that the script doesn't inadvertently open up a command or overwrite someone's file. For this to work, the script's current directory must be writable by "nobody".

#!/usr/local/bin/perl

use CGI;

$query = new CGI;

print $query->header;

print $query->start\_html("Save and Restore Example");

print "<H1>Save and Restore Example</H1>\n";

# Here's where we take action on the previous request

&save\_parameters($query) if $query->param('action') eq 'save';

$query = &restore\_parameters($query) if $query->param('action') eq 'restore';

# Here's where we create the form

print $query->startform;

print "Popup 1: ",$query->popup\_menu('popup1',['eenie','meenie','minie']),"\n";

print "Popup 2: ",$query->popup\_menu('popup2',['et','lux','perpetua']),"\n";

print "<P>";

print "Save/restore state from file: ",$query->textfield('savefile','state.sav'),"\n";

print "<P>";

print $query->submit('action','save'),$query->submit('action','restore');

print $query->submit('action','usual query');

print $query->endform;

# Here we print out a bit at the end

print $query->end\_html;

sub save\_parameters {

local($query) = @\_;

local($filename) = &clean\_name($query->param('savefile'));

if (open(FILE,">$filename")) {

$query->save(FILE);

close FILE;

print "<STRONG>State has been saved to file $filename</STRONG>\n";

} else {

print "<STRONG>Error:</STRONG> couldn't write to file $filename: $!\n";

}

}

sub restore\_parameters {

local($query) = @\_;

local($filename) = &clean\_name($query->param('savefile'));

if (open(FILE,$filename)) {

$query = new CGI(FILE); # Throw out the old query, replace it with a new one

close FILE;

print "<STRONG>State has been restored from file $filename</STRONG>\n";

} else {

print "<STRONG>Error:</STRONG> couldn't restore file $filename: $!\n";

}

return $query;

}

# Very important subroutine -- get rid of all the naughty

# metacharacters from the file name. If there are, we

# complain bitterly and die.

sub clean\_name {

local($name) = @\_;

unless ($name=~/^[\w\.\_-]+$/) {

print "<STRONG>$name has naughty characters. Only ";

print "alphanumerics are allowed. You can't use absolute names.</STRONG>";

die "Attempt to use naughty characters";

}

return $name;

}

If you use the CGI save() and restore() methods a lot, you might be interested in the *Boulderio* file format. It's a way of transferring semi-strucured data from the standard output of one program to the standard input of the next. It comes with a simple Perl database that allows you to store and retrieve records from a DBM or DB\_File database, and is compatible with the format used by save() and restore(). You can get more information on Boulderio from:

<http://stein.cshl.org/software/boulder/>

**A Script that Uses Self-Referencing URLs to Jump to Internal Links**

(Without losing form information).

Many people have experienced problems with internal links on pages that have forms. Jumping around within the document causes the state of the form to be reset. A partial solution is to use the self\_url() method to generate a link that preserves state information. This script illustrates how this works.

#!/usr/local/bin/perl

use CGI;

$query = new CGI;

# We generate a regular HTML file containing a very long list

# and a popup menu that does nothing except to show that we

# don't lose the state information.

print $query->header;

print $query->start\_html("Internal Links Example");

print "<H1>Internal Links Example</H1>\n";

print "<A NAME=\"start\"></A>\n"; # an anchor point at the top

# pick a default starting value;

$query->param('amenu','FOO1') unless $query->param('amenu');

print $query->startform;

print $query->popup\_menu('amenu',[('FOO1'..'FOO9')]);

print $query->submit,$query->endform;

# We create a long boring list for the purposes of illustration.

$myself = $query->self\_url;

print "<OL>\n";

for (1..100) {

print qq{<LI>List item #$\_<A HREF="$myself#start">Jump to top</A>\n};

}

print "</OL>\n";

print $query->end\_html;

**Multiple forms on the same page**

There's no particular trick to this. Just remember to close one form before you open another one. You can reuse the same query object or create a new one. Either technique works.

There is, however, a problem with maintaining the states of multiple forms. Because the browser only sends your script the parameters from the form in which the submit button was pressed, the state of all the other forms will be lost. One way to get around this, suggested in this example, is to use hidden fields to pass as much information as possible regardless of which form the user submits.

#!/usr/local/bin/perl

use CGI;

$query=new CGI;

print $query->header;

print $query->start\_html('Multiple forms');

print "<H1>Multiple forms</H1>\n";

# form 1

print "<HR>\n";

print $query->startform;

print $query->textfield('text1'),$query->submit('submit1');

print $query->hidden('text2'); # pass information from the other form

print $query->endform;

print "<HR>\n";

# form 2

print $query->startform;

print $query->textfield('text2'),$query->submit('submit2');

print $query->hidden('text1'); # pass information from the other form

print $query->endform;

print "<HR>\n";

print $query->end\_html;

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Subclassing CGI.pm**

CGI.pm uses various tricks to work in both an object-oriented and function-oriented fashion. It uses even more tricks to load quickly, despite the fact that it is a humungous module. These tricks may get in your way when you attempt to subclass CGI.pm.

If you use standard subclassing techniques and restrict yourself to using CGI.pm and its subclasses in the object-oriented manner, you'll have no problems. However, if you wish to use the function-oriented calls with your subclass, follow this model:

package MySubclass;

use vars qw(@ISA $VERSION);

require CGI;

@ISA = qw(CGI);

$VERSION = 1.0;

$CGI::DefaultClass = \_\_PACKAGE\_\_;

$AutoloadClass = 'CGI';

sub new {

....

}

1;

The first special trick is to set the CGI package variable $CGI::DefaultClass to the name of the module you are defining. If you are using perl 5.004 or higher, you can use the special token "\_\_PACKAGE\_\_" to retrieve the name of the current module. Otherwise, just hard code the name of the module. This variable tells CGI what type of default object to create when called in the function-oriented manner.

The second trick is to set the package variable $AutoloadClass to the string "CGI". This tells the CGI autoloader where to look for functions that are not defined. If you wish to override CGI's autoloader, set this to the name of your own package.

More information on extending CGI.pm can be found in my new book, *The Official Guide to CGI.pm*, which was published by John Wiley & Sons in April 1998. Check out the book's [Web site](http://www.wiley.com/compbooks/stein/), which contains multiple useful coding examples.

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Using CGI.pm with mod\_perl and FastCGI**

**FastCGI**

[FastCGI](http://www.fastcgi.com/) is a protocol invented by OpenMarket that markedly speeds up CGI scripts under certain circumstances. It works by opening up the script at server startup time and redirecting the script's IO to a Unix domain socket. Every time a new CGI request comes in, the script is passed new parameters to work on. This allows the script to perform all its time-consuming operations at initialization time (including loading CGI.pm!) and then respond quickly to new requests.

FastCGI modules are available for the Apache and NCSA servers as well as for OpenMarket's own server. In order to use FastCGI with Perl you have to run a specially-modified version of the Perl interpreter. Precompiled Binaries and a patch kit are all available on OpenMarket's FastCGI web site.

To use FastCGI with CGI.pm, change your scripts as follows:

**Old Script**

#!/usr/local/bin/perl

use CGI qw(:standard);

print header,

start\_html("CGI Script"),

h1("CGI Script"),

"Not much to see here",

hr,

address(a({href=>'/'},"home page"),

end\_html;

**New Script**

#!/usr/local/fcgi/bin/perl

use CGI::Fast qw(:standard);

# Do time-consuming initialization up here.

while (new CGI::Fast) {

print header,

start\_html("CGI Script"),

h1("CGI Script"),

"Not much to see here",

hr,

address(a({href=>'/'},"home page"),

end\_html;

}

That's all there is to it. The param() method, form-generation, HTML shortcuts, etc., all work the way you expect.

**mod\_perl**

[mod\_perl](http://www.perl.com/CPAN/modules/Apache/) is a module for the Apache Web server that embeds a Perl interpreter into the Web server. It can be run in either of two modes:

1. Server launches a new Perl interpreter every time it needs to interpret a Perl script. This speeds CGI scripts significantly because there's no overhead for launching a new Perl process.
2. A "fast" mode in which the server launches your script at initialization time. You can load all your favorite modules (like CGI.pm!) at initialization time, greatly speeding things up.

CGI.pm works with mod\_perl, versions 0.95 and higher. If you use Perl 5.003\_93 or higher, your scripts should run without any modifications. Users with earlier versions of Perl should use the *CGI::Apache* module instead. This example shows the change needed:

**Old Script**

#!/usr/local/bin/perl

use CGI qw(:standard);

print header,

start\_html("CGI Script"),

h1("CGI Script"),

"Not much to see here",

hr,

address(a({href=>'/'},"home page"),

end\_html;

**New Script**

#!/usr/bin/perl

use CGI::Apache qw(:standard);

print header,

start\_html("CGI Script"),

h1("CGI Script"),

"Not much to see here",

hr,

address(a({href=>'/'},"home page"),

end\_html;

}

**Configuration note:** When using CGI.pm with mod\_perl it is **not** necessary to enable the PerlSendHeader directive. This is handled automatically by CGI.pm and by Apache::Registry.

mod\_perl comes with a small wrapper library named *CGI::Switch* that selects dynamically between using CGI and CGI::Apache. This library is no longer needed. However users of CGI::Switch can continue to use it without risk. Note that the "simple" interface to the CGI.pm functions does not work with CGI::Switch. You'll have to use the object-oriented versions (or use the sfio version of Perl!)

If you use CGI.pm in many of your mod\_perl scripts, you may want to preload CGI.pm and its methods at server startup time. To do this, add the following line to httpd.conf:

PerlScript /home/httpd/conf/startup.pl

Create the file /home/httpd/conf/startup.pl and put in it all the modules you want to load. Include CGI.pm among them and call its [compile()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#compile) method to precompile its autoloaded methods.

#!/usr/local/bin/perl

use CGI ();

CGI->compile(':all');

Change the path to the startup script according to your preferences.

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Migrating from cgi-lib.pl**

To make it easier to convert older scripts that use cgi-lib.pl, CGI.pm provides a **CGI::ReadParse()** call that is compatible with cgi-lib.pl's **ReadParse()** subroutine.

When you call ReadParse(), CGI.pm creates an associative array named %in that contains the named CGI parameters. Multi-valued parameters are separated by "\0" characters in exactly the same way cgi-lib.pl does it. The function result is the number of parameters parsed. You can use this to determine whether the script is being called from a fill out form or not.

To port an old script to CGI.pm, you have to make just two changes:

**Old Script**

require "cgi-lib.pl";

ReadParse();

print "The price of your purchase is $in{price}.\n";

**New Script**

use CGI qw(:cgi-lib);

ReadParse();

print "The price of your purchase is $in{price}.\n";

Like cgi-lib's ReadParse, pass a variable *glob* in order to use a different variable than the default "%in":

ReadParse(\*Q);

@partners = split("\0",$Q{'golf\_partners'});

The associative array created by CGI::ReadParse() contains a special key 'CGI', which returns the CGI query object itself:

ReadParse();

$q = $in{CGI};

print $q->textfield(-name=>'wow',

-value=>'does this really work?');

This allows you to add the more interesting features of CGI.pm to your old scripts without rewriting them completely. As an added benefit, the **%in** variable is actually tie()'d to the CGI object. Changing the CGI object using **param()** will dynamically change **%in**, and vice-versa.

cgi-lib.pl's @in and $in variables are **not** supported. In addition, the extended version of ReadParse() that allows you to spool uploaded files to disk is not available. You are strongly encouraged to use CGI.pm's file upload interface instead.

See [cgi-lib\_porting.html](http://pages.cs.wisc.edu/~beechung/ref/cgi-lib_porting.html) for more details on porting cgi-lib.pl scripts to CGI.pm.

**Using the File Upload Feature**

The file upload feature doesn't work with every combination of browser and server. The various versions of Netscape and Internet Explorer on the Macintosh, Unix and Windows platforms don't all seem to implement file uploading in exactly the same way. I've tried to make CGI.pm work with all versions on all platforms, but I keep getting reports from people of instances that break the file upload feature.

Known problems include:

1. Large file uploads may fail when using SSL version 2.0. This affects the Netscape servers and possibly others that use the SSL library. I have received reports that WebSite Pro suffers from this problem. This is a documented bug in the Netscape implementation of SSL and not a problem with CGI.pm.
2. If you try to upload a **directory** path with Unix Netscape, the browser will hang until you hit the "stop" button. I haven't tried to figure this one out since I think it's dumb of Netscape to allow this to happen at all.
3. If you create the CGI object in one package (e.g. "main") and then obtain the filehandle in a different package (e.g. "foo"), the filehandle will be accessible through "main" but not "foo". In order to use the filehandle, try the following contortion:
4. $file = $query->param('file to upload');
5. $file = "main::$file";
6. ...

I haven't found a way to determine the correct caller in this situation. I might add a readFile() method to CGI if this problem bothers enough people.

The main technical challenge of handling file uploads is that it potentially involves sending more data to the CGI script than the script can hold in main memory. For this reason CGI.pm creates temporary files in either the /usr/tmp or the /tmp directory. These temporary files have names like CGItemp125421, and should be deleted automatically.

**Frequent Problems**

**When you run a script from the command line, it says "offline mode: enter name=value pairs on standard input". What do I do now?**

This is a prompt to enter some CGI parameters for the purposes of debugging. You can now type in some parameters like this:

first\_name=Fred

last\_name=Flintstone

city=Bedrock

End the list by typing a control-D (or control-Z on DOS/Windows systems).

If you want to run a CGI script from a script or batch file, and don't want this behavior, just pass it an empty parameter list like this:

my\_script.pl ''

This will work too on Unix systems:

my\_script.pl </dev/null

Another option is to use the "-no\_debug" pragma when you "use" CGI.pm. This will suppress command-line debugging completely:

use CGI qw/:standard -no\_debug/;

**CGI.pm breaks when you use "use integer"**

Due to problems that integer.pm has with unary negation, calls to CGI.pm that use the -arg=>value format will break if you load the integer.pm module. This is fixed in Perl 5.005\_61 and up.

A workaround is to put all arguments in quotes: '-arg'=>'value'

**You can't retrieve the name of the uploaded file using the param() method**

Most likely the remote user isn't using version 2.0 (or higher) of Netscape. Alternatively she just isn't filling in the form completely.

**When you accidentally try to upload a directory name, the browser hangs**

This seems to be a Netscape browser problem. It starts to upload junk to the script, then hangs. You can abort by hitting the "stop" button.

**You can read the name of the uploaded file, but can't retrieve the data**

First check that you've told CGI.pm to use the new [multipart/form-data](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#multipart) scheme. If it still isn't working, there may be a problem with the temporary files that CGI.pm needs to create in order to read in the (potentially very large) uploaded files. Internally, CGI.pm tries to create temporary files with names similar to CGITemp123456 in a temporary directory. To find a suitable directory it first looks for /usr/tmp and then for /tmp. If it can't find either of these directories, it tries for the current directory, which is usually the same directory that the script resides in.

If you're on a non-Unix system you may need to modify CGI.pm to point at a suitable temporary directory. This directory must be writable by the user ID under which the server runs (usually "nobody") and must have sufficient capacity to handle large file uploads. Open up CGI.pm, and find the line:

package TempFile;

foreach ('/usr/tmp','/tmp') {

do {$TMPDIRECTORY = $\_; last} if -d $\_ && -w \_;

}

Modify the foreach() line to contain a series of one or more directories to store temporary files in.

Alternatively, you can just skip the search entirely and force CGI.pm to store its temporary files in some logical location. Do this at the top of your script with a line like this one: $TempFile::TMPDIRECTORY='/WWW\_ROOT';

**On Windows Systems, the temporary file is never deleted, but hangs around in \temp, taking up space.**

Be sure to close the filehandle before your program exits. In fact, close the file as soon as you're finished with it, because the file will end up hanging around if the script later crashes.

Unix users don't have this problem, because well designed operating systems make it possible to delete a file without closing it.

**When you press the "back" button, the same page is loaded, not the previous one.**

Netscape 2.0's history list gets confused when processing multipart forms. If the script generates different pages for the form and the results, hitting the "back" button doesn't always return you to the previous page; instead Netscape reloads the current page. This happens even if you don't use an upload file field in your form.

A workaround for this is to use additional path information to trick Netscape into thinking that the form and the response have different URLs. I recommend giving each form a sequence number and bumping the sequence up by one each time the form is accessed:

my($s) = $query->path\_info=~/(\d+)/; # get sequence

$s++; #bump it up

# Trick Netscape into thinking it's loading a new script:

print $q->start\_multipart\_form(-action=>$q->script\_name . "/$s");

**You can't find the temporary file that CGI.pm creates**

You're encouraged to copy the data into your own file by reading from the file handle that CGI.pm provides you with. In the future there may be no temporary file at all, just a pipe. However, for now, if you really want to get at the temp file, you can retrieve its path using the [tmpFileName()](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html" \l "tmpfilename) method. Be sure to move the temporary file elsewhere in the file system if you don't want it to be automatically deleted when CGI.pm exits.

**Server Push**

CGI.pm provides four simple functions for producing multipart documents of the type needed to implement server push. To import these into your namespace, you must import the ":push" set. You are also advised to put the script into NPH mode and to set $| to 1 to avoid buffering problems.

Here is a simple script that demonstrates server push:

#!/usr/local/bin/perl

use CGI qw/:push -nph/;

$| = 1;

print multipart\_init(-boundary=>'----------------here we go!');

foreach (0 .. 4) {

print multipart\_start(-type=>'text/plain'),

"The current time is ",scalar(localtime),"\n";

if ($\_ < 4) {

print multipart\_end;

} else {

print multipart\_final;

}

sleep 1;

}

This script initializes server push by calling *multipart\_init()*. It then enters a loop in which it begins a new multipart section by calling *multipart\_start()*, prints the current local time, and ends a multipart section with *multipart\_end()*. It then sleeps a second, and begins again. On the final iteration, it ends the multipart section with *multipart\_final()* rather than with *multipart\_end()*.

multipart\_init()

multipart\_init(-boundary=>$boundary);

Initialize the multipart system. The -boundary argument specifies what MIME boundary string to use to separate parts of the document. If not provided, CGI.pm chooses a reasonable boundary for you.

multipart\_start()

multipart\_start(-type=>$type)

Start a new part of the multipart document using the specified MIME type. If not specified, text/html is assumed.

multipart\_end()

multipart\_end()

End a part. You must remember to call multipart\_end() once for each multipart\_start(), except at the end of the last part of the multipart document when multipart\_final() should be called instead of multipart\_end().

multipart\_final()

multipart\_final()

End all parts. You should call multipart\_final() rather than multipart\_end() at the end of the last part of the multipart document.

Users interested in server push applications should also have a look at the CGI::Push module.

Only Netscape Navigator supports server push. Internet Explorer browsers do not.

[Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Avoiding Denial of Service Attacks**

A potential problem with CGI.pm is that, by default, it attempts to process form POSTings no matter how large they are. A wily hacker could attack your site by sending a CGI script a huge POST of many megabytes. CGI.pm will attempt to read the entire POST into a variable, growing hugely in size until it runs out of memory. While the script attempts to allocate the memory the system may slow down dramatically. This is a form of denial of service attack.

Another possible attack is for the remote user to force CGI.pm to accept a huge file upload. CGI.pm will accept the upload and store it in a temporary directory even if your script doesn't expect to receive an uploaded file. CGI.pm will delete the file automatically when it terminates, but in the meantime the remote user may have filled up the server's disk space, causing problems for other programs.

The best way to avoid denial of service attacks is to limit the amount of memory, CPU time and disk space that CGI scripts can use. Some Web servers come with built-in facilities to accomplish this. In other cases, you can use the shell *limit* or *ulimit* commands to put ceilings on CGI resource usage.

CGI.pm also has some simple built-in protections against denial of service attacks, but you must activate them before you can use them. These take the form of two global variables in the CGI name space:

**$CGI::POST\_MAX**

If set to a non-negative integer, this variable puts a ceiling on the size of POSTings, in bytes. If CGI.pm detects a POST that is greater than the ceiling, it will immediately exit with an error message. This value will affect both ordinary POSTs and multipart POSTs, meaning that it limits the maximum size of file uploads as well. You should set this to a reasonably high value, such as 1 megabyte.

**$CGI::DISABLE\_UPLOADS**

If set to a non-zero value, this will disable file uploads completely. Other fill-out form values will work as usual.

You can use these variables in either of two ways.

1. On a script-by-script basis. Set the variable at the top of the script, right after the "use" statement:
2. use CGI qw/:standard/;
3. use CGI::Carp 'fatalsToBrowser';
4. $CGI::POST\_MAX=1024 \* 100; # max 100K posts
5. $CGI::DISABLE\_UPLOADS = 1; # no uploads

1. Globally for all scripts. Open up CGI.pm, find the definitions for $POST\_MAX and $DISABLE\_UPLOADS, and set them to the desired values. You'll find them towards the top of the file in a subroutine named initialize\_globals.

Since an attempt to send a POST larger than $POST\_MAX bytes will cause a fatal error, you might want to use CGI::Carp to echo the fatal error message to the browser window as shown in the example above. Otherwise the remote user will see only a generic "Internal Server" error message. See the manual page for CGI::Carp for more details.

An attempt to send a POST larger than $POST\_MAX bytes will cause **param()** to return an empty CGI parameter list. You can test for this event by checking **cgi\_error()**, either after you create the CGI object or, if you are using the function-oriented interface, call **param()** for the first time. If the POST was intercepted, then cgi\_error() will return the message "413 POST too large".

This error message is actually defined by the HTTP protocol, and is designed to be returned to the browser as the CGI script's status code. For example:

$uploaded\_file = param('upload');

if (!$uploaded\_file && cgi\_error()) {

print header(-status=>cgi\_error());

exit 0;

}

Some browsers may not know what to do with this status code. It may be better just to create an HTML page that warns the user of the problem. [Table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Using CGI.pm on non-Unix Platforms**

I don't have access to all the combinations of hardware and software that I really need to make sure that CGI.pm works consistently for all Web servers, so I rely heavily on helpful reports from users like yourself.

There are a number of differences in file name and text processing conventions on different platforms. By default, CGI.pm is set up to work properly on a Unix (or Linux) system. During load, it will attempt to guess the correct operating system using the Config module. Currently it guesses correctly; however if the operating system names change it may not work right. The main symptom will be that file upload does not work correctly. If this happens, find the place at the top of the script where the OS is defined, and uncomment the correct definition:

# CHANGE THIS VARIABLE FOR YOUR OPERATING SYSTEM

# $OS = 'UNIX';

# $OS = 'MACINTOSH';

# $OS = 'WINDOWS';

# $OS = 'VMS';

Other notes follow:

**Windows NT**

CGI.pm works well with WebSite, the EMWACS server, Purveyor and the Microsoft IIS server. CGI.pm must be put in the perl5 library directory, and all CGI scripts that use it should be placed in cgi-bin directory. You also need to associate the .pl suffix with perl5 using the NT file manager (Website, Purveyor), or install the correct script mapping registry keys for IIS. Perl for Windows is available from the ActiveState company, which can be found at:

<http://www.activestate.com/>

WebSite uses a slightly different cgi-bin directory structure than the standard. For this server, place the scripts in the cgi-shl directory. CGI.pm appears to work correctly in both the Windows95 and WindowsNT versions of WebSite.

Old Netscape Communications Server technical notes recommended placing perl.exe in cgi-bin. This a very bad idea because it opens up a gaping security hole. Put a C .exe wrapper around the perl script until such time as Netscape recognizes NT file manager associations, or provides a Perl-compatible DLL library for its servers.

If you find that binary files get slightly larger when uploaded but that text files remain the same, then binary made is not correctly activated. Be sure to set the $OS variable to 'NT' or 'WINDOWS'. If you continue to have problems, make sure you're calling **binmode()** on the filehandle that you use to write the uploaded file to disk.

**VMS**

I don't have access to a VMS machine, and I'm not sure whether file upload works correctly. Other features are known to work.

**Macintosh**

Most CGI.pm features work with MacPerl version 5.0.6r1 or higher under the WebStar and MacHTTP servers. In order to install a Perl program to use with the Web, you'll need Matthias Nuuracher's PCGI extension, available at:

<ftp://err.ethz.ch/pub/neeri/MacPerl/>

Known incompatibilities between CGI.pm and MacPerl include:

1. The perl compiler will object to the use of -values in named parameters. Put single quotes around this parameter ('-values') or use the singular form ('-value') instead.
2. File upload isn't working in my hands (Perl goes into an endless loop). Other people have gotten it to work.

**The Relation of this Library to the CGI Modules**

This library is maintained in parallel with the full featured CGI, URL, and HTML modules. I use this library to test out new ideas before incorporating them into the CGI hierarchy. I am continuing to maintain and improve this library in order to satisfy people who are looking for an easy-to-use introduction to the world of CGI scripting.

The CGI::\* modules are being reworked to be interoperable with the excellent LWP modules. Stay tuned.

The current version of CGI.pm can be found at:

[http://www.genome.wi.mit.edu/ftp/pub/software/WWW/](http://www.genome.wi.mit.edu/ftp/pub/software/WWW)

You are encouraged to look at these other Web-related modules:

[CGI::Base,CGI::Form,CGI::MiniSrv,CGI::Request and CGI::URI::URL](http://www.genome.wi.mit.edu/ftp/pub/software/WWW/CGIperl/)

Modules for parsing script input, manipulating URLs, creating forms and even launching a miniature Web server.

[libwww-perl](http://www.ics.uci.edu/pub/websoft/libwww-perl/)

Modules for fetching Web resources from within Perl, writing Web robots, and much more.

You might also be interested in two packages for creating graphics on the fly:

[GD.html](http://www.genome.wi.mit.edu/ftp/pub/software/WWW/GD.html)

A module for creating GIF images on the fly, using Tom Boutell's [gd](http://www.boutell.com/gd/) graphics library.

[qd.pl](http://www.genome.wi.mit.edu/ftp/pub/software/utilities/)

A library for creating Macintosh PICT files on the fly (which can be converted to GIF or JPEG using NetPBM).

For a collection of CGI scripts of various levels of complexity, see the companion pages for my book [How to Set Up and Maintain a World Wide Web Site](http://www.genome.wi.mit.edu/WWW/)

**Distribution Information:**

This code is copyright 1995-1998 by Lincoln Stein. It may be used and modified freely, but I do request that this copyright notice remain attached to the file. You may modify this module as you wish, but if you redistribute a modified version, please attach a note listing the modifications you have made.

**The CGI.pm Book**

*The Official Guide to CGI.pm*, by Lincoln Stein, is packed with tips and techniques for using the module, along with information about the module's internals that can't be found anywhere else. It is available on bookshelves now, or can be ordered from [amazon.com](http://www.amazon.com/). Also check the book's companion Web site at:

<http://www.wiley.com/compbooks/stein/>

**CGI.pm and the Year 2000 Problem**

Versions of CGI.pm prior to 2.36 suffered a year 2000 problem in the handling of cookies. Cookie expiration dates were expressed using two digits as dictated by the then-current Netscape cookie protocol. The cookie protocol has since been cleaned up. My belief is that versions of CGI.pm 2.36 and higher are year 2000 compliant.

**The CGI-perl mailing list**

The CGI Perl mailing list is defunct and is unlikely to be resurrected. Please address your questions to [comp.infosystems.www.authoring.cgi](news:comp.infosystems.www.authoring.cgi) if they relate to the CGI protocol or the usage of CGI.pm *per gse*, or to [comp.lang.perl.misc](news:comp.lang.perl.misc) for Perl **language** issues. Please read this documentation thoroughly, read the FAQs for these newsgroups and scan through previous messages before you make a posting. Respondents are not always friendly to people who neglect to do so!

**Bug Reports**

Address bug reports and comments to:  
[lstein@cshl.org](mailto:lstein@cshl.org). When sending bug reports, please provide the following information:

* the version of CGI.pm (perl -MCGI -e 'print $CGI::VERSION')
* the version of Perl (perl -v)
* the name and version of your Web server
* the name and version of the operating system you are using
* if applicable, the name and version of the browser you are using
* a short test script that reproduces the problem (30 lines or less)

It is very important that I receive this information in order to help you.

[Up to table of contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

**Revision History**

**Version 2.98**

1. Fixed crash in Dump() function.

**Version 2.97**

1. Sigh. Uploaded wrong 2.96 to CPAN.

**Version 2.96**

1. More bugfixes to the -style argument.

**Version 2.95**

1. Fixed bugs in start\_html(-style=>...) support introduced in 2.94.

**Version 2.94**

1. Removed warning from reset() method.
2. Moved and tags into the :html3 group. Hope this removes undefined CGI::Area errors.
3. Changed CGI::Carp to play with mod\_perl2 and to (hopefully) restore reporting of compile-time errors.
4. Fixed potential deadlock between web server and CGI.pm when aborting a read due to POST\_MAX (reported by Antti Lankila).
5. Fixed issue with tag-generating function not incorporating content when first variable undef.
6. Fixed cross-site scripting bug reported by obscure.
7. Fixed Dump() function to return correctly formed XHTML - bug reported by Ralph Siemsen.

**Version 2.93**

1. Fixed embarassing bug in mp1 support.

**Version 2.92**

1. Fix to be P3P compliant submitted from MPREWITT.
2. Added CGI->r() API for mod\_perl1/mod\_perl2.
3. Fixed bug in redirect() that was corrupting cookies.
4. Minor fix to behavior of reset() button to make it consistent with submit() button (first time this has been changed in 9 years).
5. Patch from Dan Kogai to handle UTF-8 correctly in 5.8 and higher.
6. Patch from Steve Hay to make CGI::Carp's error messages appear on MSIE browsers.
7. Added Yair Lenga's patch for non-urlencoded postings.
8. Added Stas Bekman's patches for mod\_perl 2 compatibility.
9. Fixed uninitialized escape behavior submitted by William Campbell.
10. Fixed tied behavior so that you can pass arguments to tie()
11. Fixed incorrect generation of URLs when the path\_info contains + and other odd characters.
12. Fixed redirect(-cookies=>$cookie) problem.
13. Fixed tag generation bug that affects -javascript passed to start\_html().

**Version 2.91**

1. Attribute generation now correctly respects the value of autoEscape().
2. Fixed endofrm() syntax error introduced by Ben Edgington's patch.

**Version 2.90**

1. Fixed bug in redirect header handling.
2. Added P3P option to header().
3. Patches from Alexey Mahotkin to make CGI::Carp work correctly with object-oriented exceptions.
4. Removed inaccurate description of how to set multiple cookies from CGI::Cookie pod file.
5. Patch from Kevin Mahony to prevent running out of filehandles when uploading lots of files.
6. Documentation enhancement from Mark Fisher to note that the import\_names() method transforms the parameter names into valid Perl names.
7. Patch from Dan Harkless to suppress lang attribute in <html> tag if specified as a null string.
8. Patch from Ben Edgington to fix broken XHTML-transitional 1.0 validation on endform().
9. Custom html header fix from Steffen Beyer (first letter correctly upcased now)
10. Added a -verbatim option to stylesheet generation from Michael Dickson
11. Faster delete() method from Neelam Gupta
12. Fixed broken Cygwin support.
13. Added empty charset support from Bradley Baetz
14. Patches from Doug Perham and Kevin Mahoney to fix file upload failures when uploaded file is a multiple of 4096.

**Version 2.89**

1. Fixed behavior of ACTION tag when POSTING to a URL that has a query string.
2. Added Patch from Michael Rommel to handle multipart/mixed uploads from Opera

**Version 2.88**

1. Fixed problem with uploads being refused under Perl 5.8 when under Taint mode.
2. Fixed uninitialized variable warnings under Perl 5.8.
3. Fixed CGI::Pretty regression test failures.

**Version 2.87**

1. Security hole patched: when processing multipart/form-data postings, most arguments were being untainted silently. Returned arguments are now tainted correctly. This may cause some scripts to fail that used to work (thanks to Nick Cleaton for pointing this out and persisting until it was fixed).
2. Update for mod\_perl 2.0.
3. Pragmas such as -no\_xhtml are now respected in mod\_perl environment.

**Version 2.86**

1. Fixes for broken CGI::Cookie expiration dates introduced in 2.84.

**Version 2.85**

1. Fix for broken autoEscape function introduced in 2.84.

**Version 2.84**

1. Fix for failed file uploads on Cygwin platforms.
2. HTML escaping code now replaced 0x8b and 0x9b with unicode references ‹ and \*#8250;

**Version 2.83**

1. Fixed autoEscape() documentation inconsistencies.
2. Patch from Ville Skyttä to fix a number of XHTML inconsistencies.
3. Added Max-Age to list of CGI::Cookie headers.

**Version 2.82**

1. Patch from Rudolf Troller to add attribute setting and option groups to form fields.
2. Patch from Simon Perreault for silent crashes when using CGI::Carp under mod\_perl.
3. Patch from Scott Gifford allows you to set the program name for CGI::Carp.

**Version 2.81**

1. Removed extraneous slash from end of stylesheet tags generated by start\_html in non-XHTML mode.
2. Changed behavior of CGI::Carp with respect to eval{} contexts so that output behaves properly in mod\_perl environments.
3. Fixed default DTD so that it validates with W3C validator.

**Version 2.80**

1. Fixed broken messages in CGI::Carp.
2. Changed checked="1" to checked="checked" for real XHTML compatibility.
3. Resurrected REQUEST\_URI code so that url() works correctly with multiviews.

**Version 2.79**

1. Changes to CGI::Carp to avoid "subroutine redefined" error messages.
2. Default DTD is now XHTML 1.0 Transitional
3. Patches to support all HTML4 tags.

**Version 2.78**

1. Added ability to change encoding in <?xml> assertion.
2. Fixed the old escapeHTML('CGI') ne "CGI" bug
3. In accordance with XHTML requirements, there are no longer any minimized attributes, such as "checked".
4. Patched bug which caused file uploads of exactly 4096 bytes to be truncated to 4094 (thanks to Kevin Mahony)
5. New tests and fixes to CGI::Pretty (thanks to Michael Schwern).

**Version 2.77**

1. No new features, but released in order to fix an apparent CPAN bug.

**Version 2.76**

1. New esc.t regression test for EBCDIC translations courtesy Peter Prymmer.
2. Patches from James Jurach to make compatible with FCGI-ProcManager
3. Additional fields passed to header() (like -Content\_disposition) now honor initial capitalization.
4. Patch from Andrew McNaughton to handle utf-8 escapes (%uXXXX codes) in URLs.

**Version 2.752**

1. Syntax error in the autoloaded Fh::new() subroutine.
2. Better error reporting in autoloaded functions.

**Version 2.751**

1. Tiny tweak to filename regular expression function on line 3355.

**Version 2.75**

1. Fixed bug in server push boundary strings (CGI.pm and CGI::Push).
2. Fixed bug that occurs when uploading files with funny characters in the name
3. Fixed non-XHTML-compliant attributes produced by textfield()
4. Added EPOC support, courtesy Olaf Flebbe
5. Fixed minor XHTML bugs.
6. Made escape() and unescape() symmetric with respect to EBCDIC, courtesy Roca, Ignasi <ignasi.roca@fujitsu.siemens.es>
7. Removed uninitialized variable warning from CGI::Cookie, provided by Atipat Rojnuckarin <rojnuca@yahoo.com>
8. Fixed bug in CGI::Pretty that causes it to print partial end tags when the $INDENT global is changed.
9. Single quotes are changed to character entity ' for compatibility with URLs.

**Version 2.74**

September 13, 2000

1. Quashed one-character bug that caused CGI.pm to fail on file uploads.

**Version 2.73**

September 12, 2000

1. Added -base to the list of arguments accepted by url().
2. Fixes to XHTML support.
3. POST parameters no longer show up in the Location box.

**Version 2.72**

August 19, 2000

1. Fixed the defaults button so that it works again
2. Charset is now correctly saved and restored when saving to files
3. url() now works correctly when given scripts with %20 and other escapes in the additional path info. This undoes a patch introduced in version 2.47 that I no longer understand the rationale for.

**Version 2.71**

August 13, 2000

1. Newlines in the value attributes of hidden fields and other form elements are now escaped when using ISO-Latin.
2. Inline script and style sections are now protected as CDATA sections when XHTML mode is on (the default).

**Version 2.70**

August 4, 2000

1. Fixed bug in scrolling\_list() which omitted a space in front of the "multiple" attribute.
2. Squashed the "useless use of string in void context" message from redirects.

**Version 2.69**

1. startform() now creates default ACTION for POSTs as well as GETs. This may break some browsers, but it no longer violates the HTML spec.
2. CGI.pm now emits XHTML by default. Disable with -no\_xhtml.
3. We no longer interpret &#ddd sequences in non-latin character sets.

**Version 2.68**

1. No longer attempts to escape characters when dealing with non ISO-8861 character sets.
2. checkbox() function now defaults to using -value as its label, rather than -name. The current behavior is what has been documented from the beginning.
3. -style accepts array reference to incorporate multiple stylesheets into document.
4. Fixed two bugs that caused the -compile pragma to fail with a syntax error.

**Version 2.67**

1. Added XHTML support (incomplete; tags need to be lowercased).
2. Fixed CGI/Carp when running under mod\_perl. Probably broke in other contexts.
3. Fixed problems when passing multiple cookies.
4. Suppress warnings from \_tableize() that were appearing when using -w switch with radio\_group() and checkbox\_group().
5. Support for the header() -attachment argument, which can give pages a default file name when saving to disk.

**Version 2.66**

1. 2.65 changes in make\_attributes() broke HTTP header functions (including redirect), so made it context sensitive.

**Version 2.65**

1. Fixed regression tests to skip tests that require implicit fork on machines without fork().
2. Changed make\_attributes() to automatically escape any HTML reserved characters.
3. Minor documentation fix in javascript example.

**Version 2.64**

1. Changes introduced in 2.63 broke param() when retrieving parameter lists containing only a single argument. This is now fixed.
2. self\_url() now defaults to returning parameters delimited with semicolon. Use the pragma -oldstyle\_urls to get the old "&" delimiter.

**Version 2.63**

1. Fixed CGI::Push to pull out parameters correctly.
2. Fixed redirect() so that it works with default character set
3. Changed param() so as to returned empty string '' when referring to variables passed in query strings like 'name1=&name2'

**Version 2.62**

1. Fixed broken ReadParse() function, and added regression tests
2. Fixed broken CGI::Pretty, and added regression tests

**Version 2.61**

1. Moved more functions from CGI.pm proper into CGI/Util.pm. CGI/Cookie should now be standalone.
2. Disabled per-user temporary directories, which were causing grief.

**Version 2.60**

1. Fixed junk appearing in autogenerated HTML functions when using object-oriented mode.

**Version 2.59**

1. autoescape functionality breaks too much existing code, removed it.
2. use escapeHTML() manually

**Version 2.58**

This is the release version of 2.57.

**Version 2.57**

1. Added -debug pragma and turned off auto reading of STDIN.
2. Default DTD updated to HTML 4.01 transitional.
3. Added charset() method and the -charset argument to header().
4. Fixed behavior of escapeHTML() to respect charset() and to escape nasty Windows characters (thanks to Tom Christiansen).
5. Handle REDIRECT\_QUERY\_STRING correctly.
6. Removed use\_named\_parameters() because of dependency problems and general lameness.
7. Fixed problems with bad HREF links generated by url(-relative=>1) when the url is like /people/.
8. Silenced a warning on upload (patch provided by Jonas Liljegren)
9. Fixed race condition in CGI::Carp when errors occur during parsing (patch provided by Maurice Aubrey).
10. Fixed failure of url(-path\_info=>1) when path contains % signs.
11. Fixed warning from CGI::Cookie when receiving foreign cookies that don't use name=value format.
12. Fixed incompatibilities with file uploading on VMS systems.

**Version 2.56**

1. Fixed bugs in file upload introduced in version 2.55
2. Fixed long-standing bug that prevented two files with identical names from being uploaded.

**Version 2.55**

1. Fixed cookie regression test so as not to produce an error.
2. Fixed path\_info() and self\_url() to work correctly together when path\_info() modified.
3. Removed manify warnings from CGI::{Switch,Apache}.

**Version 2.54**

1. This will be the last release of the monolithic CGI.pm module. Later versions will be modularized and optimized.
2. DOMAIN tag no longer added to cookies by default. This will break some versions of Internet Explorer, but will avoid breaking networks which use host tables without fully qualified domain names. For compatibility, please always add the -domain tag when creating cookies.
3. Fixed escape() method so that +'s are treated correctly.
4. Updated CGI::Pretty module.

**Version 2.53**

1. Forgot to upgrade regression tests before releasing 2.52. **NOTHING ELSE HAS CHANGED IN LIBRARY**

**Version 2.52**

1. Spurious newline in checkbox() routine removed. (courtesy John Essen)
2. TEXTAREA linebreaks now respected in dump() routine. (courtesy John Essen)
3. Patches for DOS ports (courtesy Robert Davies)
4. Patches for VMS
5. More fixes for cookie problems
6. Fix CGI::Carp so that it doesn't affect eval{} blocks (courtesy Byron Brummer)

**Version 2.51**

1. Fixed problems with cookies not being remembered when sent to IE 5.0 (and Netscape 5.0 too?)
2. Numerous HTML compliance problems in cgi\_docs.html; fixed thanks to Michael Leahy

**Version 2.50**

1. Added a new Vars() method to retrieve all parameters as a tied hash.
2. Untainted tainted tempfile name so that script doesn't fail on terminal unlink.
3. Made picking of upload tempfile name more intelligent so that doesn't fail in case of name collision.
4. Fixed handling of expire times when passed an absolute timestamp.
5. Changed dump() to Dump() to avoid name clashes.

**Version 2.49**

1. Fixes for FastCGI (globals not getting reset)
2. Fixed url() to correctly handle query string and path under MOD\_PERL

**Version 2.48**

1. Reverted detection of MOD\_PERL to avoid breaking PerlEX.

**Version 2.47**

1. Patch to fix file upload bug appearing in IE 3.01 for Macintosh/PowerPC.
2. Replaced use of $ENV{SCRIPT\_NAME} with $ENV{REQUEST\_URI} when running under Apache, to fix self-referencing URIs.
3. Fixed bug in escapeHTML() which caused certain constructs, such as CGI->image\_button(), to fail.
4. Fixed bug which caused strong('CGI') to fail. Be careful to use CGI::strong('CGI') and not CGI->strong('CGI'). The latter will produce confusing results.
5. Added **upload()** function, as a preferred replacement for the "filehandle as string" feature.
6. Added **cgi\_error()** function.
7. Rewrote file upload handling to return undef rather than dieing when an error is encountered. Be sure to call **cgi\_error()** to find out what went wrong.

**Version 2.46**

1. Fix for failure of the "include" tests under mod\_perl
2. Added end\_multipart\_form to prevent failures during qw(-compile :all)

**Version 2.45**

1. Multiple small documentation fixes
2. *CGI::Pretty* didn't get into 2.44. Fixed now.

**Version 2.44**

1. Fixed file descriptor leak in upload function.
2. Fixed bug in header() that prevented fields from containing double quotes.
3. Added Brian Paulsen's *CGI::Pretty* package for pretty-printing output HTML.
4. Removed CGI::Apache and CGI::Switch from the distribution.
5. Generated start\_\* shortcuts so that start\_table(), end\_table(), start\_ol(), end\_ol(), and so forth now work (see the docs on how to enable this feature).
6. Changed accept() to Accept(), sub() to Sub(). There's still a conflict with reset(), but this will break too many existing scripts!

**Version 2.43**

1. Fixed problem with "use strict" and file uploads (thanks to Peter Haworth)
2. Fixed problem with not MSIE 3.01 for the power\_mac not doing file uploads right.
3. Fixed problem with file upload on IIS 4.0 when authorization in use.
4. -content\_type and '-content-type' can now be provided to header() as synonyms for -type.
5. CGI::Carp now escapes the ampersand BEFORE escaping the > and < signs.
6. Fixed "not an array reference" error when passing a hash reference to radio\_group().
7. Fixed non-removal of uploaded TMP files on NT platforms which occurs when server runs on non-C drive (thanks to Steve Kilbane for finding this one).

**Version 2.42**

1. Too many screams of anguish at changed behavior of url(). Is now back to its old behavior by default, with options to generate all the variants.
2. Added regression tests. "make test" now works.
3. Documentation fixes.
4. Fixes for Macintosh uploads, but uploads STILL do not work pending changes to MacPerl.

**Version 2.41**

1. url() method now includes the path info. Use script\_name() to get it without path info().
2. Changed handling of empty attributes in HTML tag generation. Be warned! Use table({-border=>undef}) rather than table({-border=>''}).
3. Changes to allow uploaded filenames to be compared to other strings with "eq", "cmp" and "ne".
4. Changes to allow CGI.pm to coexist more peacefully with ActiveState PerlEX.
5. Changes to prevent exported variables from clashing when importing ":all" set in combination with cookies.

**Version 2.40**

1. CGI::Carp patched to work better with mod\_perl (thanks to Chris Dean).
2. Uploads of files whose names begin with numbers or the Windows \\UNC\shared\file nomenclature should no longer fail.
3. The <STYLE> tag (for cascading style sheets) now generates the required TYPE attribute.
4. Server push primitives added, thanks to Ed Jordan.
5. Table and other HTML3 functions are now part of the :standard set.
6. Small documentation fixes.

*TO DO:*

1. Do something about the DTD mess. The module should generate correct DTDs, or at least offer the programmer a way to specify the correct one.
2. Split CGI.pm into CGI processing and HTML-generating modules.
3. More robust file upload (?still not working on the Macintosh?).
4. Bring in all the HTML4 functionality, particular the accessibility features.

**Version 2.39**

1. file uploads failing because of VMS patch; fixed.
2. -dtd parameter was not being properly processed.

**Version 2.38**

I finally got tired of all the 2.37 betas and released 2.38. The main difference between this version and the last 2.37 beta (2.37b30) are some fixes for VMS. This should allow file upload to work properly on all VMS Web servers.

**Version 2.37, various beta versions**

1. Added a CGI::Cookie::parse() method for lucky mod\_perl users.
2. No longer need separate -values and -labels arguments for multi-valued form elements.
3. Added better interface to raw cookies (fix courtesy Ken Fox, kfox@ford.com)
4. Added param\_fetch() function for direct access to parameter list.
5. Fix to checkbox() to allow for multi-valued single checkboxes (weird problem).
6. Added a compile() method for those who want to compile without importing.
7. Documented the import pragmas a little better.
8. Added a -compile switch to the use clause for the long-suffering mod\_perl and Perl compiler users.
9. Fixed initialization routines so that FileHandle and type globs work correctly (and hash initialization doesn't fail!).
10. Better deletion of temporary files on NT systems.
11. Added documentation on escape(), unescape(), unescapeHTML() and unescapeHTML() subroutines.
12. Added documentation on creating subclasses.
13. Fixed problem when calling $self->SUPER::foo() from inheriting subclasses.
14. Fixed problem using filehandles from within subroutines.
15. Fixed inability to use the string "CGI" as a parameter.
16. Fixed exponentially growing $FILLUNIT bug
17. Check for undef filehandle in read\_from\_client()
18. Now requires the UNIVERSAL.pm module, present in Perl 5.003\_7 or higher.
19. Fixed problem with uppercase-only parameters being ignored.
20. Fixed vanishing cookie problem.
21. Fixed warning in initialize\_globals() under mod\_perl.
22. File uploads from Macintosh versions of MSIE should now work.
23. Pragmas now preceded by dashes (-nph) rather than colons (:nph). Old style is supported for backward compatability.
24. Can now pass arguments to all functions using {} brackets, resolving historical inconsistencies.
25. Removed autoloader warnings about absent MultipartBuffer::DESTROY.
26. Fixed non-sticky checkbox() when -name used without -value.
27. Hack to fix path\_info() in IIS 2.0. Doesn't help with IIS 3.0.
28. Parameter syntax for debugging from command line now more straightforward.
29. Added $DISABLE\_UPLOAD to disable file uploads.
30. Added $POST\_MAX to error out if POSTings exceed some ceiling.
31. Fixed url\_param(), which wasn't working at all.
32. Fixed variable suicide problem in s///e expressions, where the autoloader was needed during evaluation.
33. Removed excess spaces between elements of checkbox and radio groups
34. Can now create "valueless" submit buttons
35. Can now set path\_info as well as read it.
36. ReadParse() now returns a useful function result.
37. import\_names() now allows you to optionally clear out the namespace before importing (for mod\_perl users)
38. Made it possible to have a popup menu or radio button with a value of "0".
39. link() changed to Link() to avoid overriding native link function.
40. Takes advantage of mod\_perl's register\_cleanup() function to clear globals.
41. <LAYER> and <ILAYER> added to :html3 functions.
42. Fixed problems with private tempfiles and NT/IIS systems.
43. No longer prints the DTD by default (I bet no one will complain).
44. Allow underscores to replace internal hyphens in parameter names.
45. CGI::Push supports heterogeneous MIME types and adjustable delays between pages.
46. url\_param() method added for retrieving URL parameters even when a fill-out form is POSTed.
47. Got rid of warnings when radio\_group() is called.
48. Cookies now moved to their very own module.
49. Fixed documentation bug in CGI::Fast.
50. Added a :no\_debug pragma to the import list.

**Version 2.36**

1. Expanded JavaScript functionality
2. Preliminary support for cascading stylesheets
3. Security fixes for file uploads:
   * Module will bail out if its temporary file already exists
   * Temporary files can now be made completely private to avoid peeking by other users or CGI scripts.
4. *use CGI qw/:nph/* wasn't working correctly. Now it is.
5. Cookie and HTTP date formats didn't meet spec. Thanks to Mark Fisher (fisherm@indy.tce.com) for catching and fixing this.

p

**Version 2.35**

1. Robustified multipart file upload against incorrect syntax in POST.
2. Fixed more problems with mod\_perl.
3. Added -noScript parameter to start\_html().
4. Documentation fixes.

**Version 2.34**

1. Stupid typo fix

**Version 2.33**

1. Fixed a warning about an undefined environment variable.
2. Doug's patch for redirect() under mod\_perl
3. Partial fix for busted inheritence from CGI::Apache
4. Documentation fixes.

**Version 2.32**

1. Improved support for Apache's mod\_perl.
2. Changes to better support inheritance.
3. Support for OS/2.

**Version 2.31**

1. New **uploadInfo()** method to obtain header information from uploaded files.
2. **cookie()** without any arguments returns all the cookies passed to a script.
3. Removed annoying warnings about $ENV{NPH} when running with the -w switch.
4. Removed operator overloading throughout to make compatible with new versions of perl.
5. **-expires** now implies the **-date** header, to avoid clock skew.
6. WebSite passes cookies in $ENV{COOKIE} rather than $ENV{HTTP\_COOKIE}. We now handle this, even though it's O'Reilly's fault.
7. Tested successfully against new sfio I/O layer.
8. Documentation fixes.

**Version 2.30**

1. Automatic detection of operating system at load time.
2. Changed select() function to Select() in order to avoid conflict with Perl built-in.
3. Added Tr() as an alternative to TR(); some people think it looks better that way.
4. Fixed problem with autoloading of MultipartBuffer::DESTROY code.
5. Added the following methods:
   * virtual\_host()
   * server\_software()
6. Automatic NPH mode when running under Microsoft IIS server.

**Version 2.29**

1. Fixed cookie bugs
2. Fixed problems that cropped up when useNamedParameters was set to 1.
3. Prevent CGI::Carp::fatalsToBrowser() from crapping out when encountering a die() within an eval().
4. Fixed problems with filehandle initializers.

**Version 2.28**

1. Added support for NPH scripts; also fixes problems with Microsoft IIS.
2. Fixed a problem with checkbox() values not being correctly saved and restored.
3. Fixed a bug in which CGI objects created with empty string initializers took on default values from earlier CGI objects.
4. Documentation fixes.

**Version 2.27**

1. Small but important bug fix: the automatic capitalization of tag attributes was accidentally capitalizing the VALUES as well as the ATTRIBUTE names (oops).

**Version 2.26**

1. Changed behavior of scrolling\_list(), checkbox() and checkbox\_group() methods so that defaults are honored correctly. The "fix" causes endform() to generate additional <INPUT TYPE="HIDDEN"> tags -- don't be surpised.
2. Fixed bug involving the detection of the SSL protocol.
3. Fixed documentation error in position of the -meta argument in start\_html().
4. HTML shortcuts now generate tags in ALL UPPERCASE.
5. start\_html() now generates correct SGML header:
6. <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">

1. CGI::Carp no longer fails "use strict refs" pragma.

**Version 2.25**

1. Fixed bug that caused bad redirection on destination URLs with arguments.
2. Fixed bug involving use\_named\_parameters() followed by start\_multipart\_form()
3. Fixed bug that caused incorrect determination of binmode for Macintosh.
4. Spelling fixes on documentation.

**Version 2.24**

1. Fixed bug that caused generation of lousy HTML for some form elements
2. Fixed uploading bug in Windows NT
3. Some code cleanup (not enough)

**Version 2.23**

1. Fixed an obscure bug that caused scripts to fail mysteriously.
2. Fixed auto-caching bug.
3. Fixed bug that prevented HTML shortcuts from passing taint checks.
4. Fixed some -w warning problems.

**Version 2.22**

1. New CGI::Fast module for use with FastCGI protocol. See pod documentation for details.
2. Fixed problems with inheritance and autoloading.
3. Added TR() (<tr>) and PARAM() (<param>) methods to list of exported HTML tag-generating functions.
4. Moved all CGI-related I/O to a bottleneck method so that this can be overridden more easily in mod\_perl (thanks to Doug MacEachern).
5. put() method as substitute for print() for use in mod\_perl.
6. Fixed crash in tmpFileName() method.
7. Added tmpFileName(), startform() and endform() to export list.
8. Fixed problems with attributes in HTML shortcuts.
9. Functions that don't actually need access to the CGI object now no longer generate a default one. May speed things up slightly.
10. Aesthetic improvements in generated HTML.
11. New examples.

**Version 2.21**

1. Added the *-meta* argument to *start\_html()*.
2. Fixed hidden fields (again).
3. Radio\_group() and checkbox\_group() now return an appropriate scalar value when called in a scalar context, rather than returning a numeric value!
4. Cleaned up the formatting of form elements to avoid unesthetic extra spaces within the attributes.
5. HTML elements now correctly include the closing tag when parameters are present but null: em('')
6. Added password\_field() to the export list.

**Version 2.20**

1. Dumped the SelfLoader because of problems with running with taint checks and rolled my own. Performance is now significantly improved.
2. Added HTML shortcuts.
3. *import()* now adheres to the Perl module conventions, allowing CGI.pm to import any or all method names into the user's name space.
4. Added the ability to initialize CGI objects from strings and associative arrays.
5. Made it possible to initialize CGI objects with filehandle references rather than filehandle strings.
6. Added the delete\_all() and append() methods.
7. CGI objects correctly initialize from filehandles on NT/95 systems now.
8. Fixed the problem with binary file uploads on NT/95 systems.
9. Fixed bug in redirect().
10. Added '-Window-target' parameter to redirect().
11. Fixed import\_names() so that parameter names containing funny characters work.
12. Broke the unfortunate connection between cookie and CGI parameter name space.
13. Fixed problems with hidden fields whose values are 0.
14. Cleaned up the documentation somewhat.

**Version 2.19**

1. Added cookie() support routines.
2. Added -expires parameter to header().
3. Added cgi-lib.pl compatability mode.
4. Made the module more configurable for different operating systems.
5. Fixed a dumb bug in JavaScript button() method.

**Version 2.18**

1. Fixed a bug that corrects a hang that occurs on some platforms when processing file uploads. Unfortunately this disables the check for bad Netscape uploads.
2. Fixed bizarre problem involving the inability to process uploaded files that begin with a non alphabetic character in the file name.
3. Fixed a bug in the hidden fields involving the -override directive being ignored when scalar defaults were passed.
4. Added documentation on how to disable the SelfLoader features.

**Version 2.17**

1. Added support for the SelfLoader module.
2. Added oodles of JavaScript support routines.
3. Fixed bad bug in query\_string() method that caused some parameters to be silently dropped.
4. Robustified file upload code to handle premature termination by the client.
5. Exported temporary file names on file upload.
6. Removed spurious "uninitialized variable" warnings that appeared when running under 5.002.
7. Added the Carp.pm library to the standard distribution.
8. Fixed a number of errors in this documentation, and probably added a few more.
9. Checkbox\_group() and radio\_group() now return the buttons as arrays, so that you can incorporate the individual buttons into specialized tables.
10. Added the '-nolabels' option to checkbox\_group() and radio\_group(). Probably should be added to all the other HTML-generating routines.
11. Added the url() method to recover the URL without the entire query string appended.
12. Added request\_method() to list of environment variables available.
13. Would you believe it? Fixed hidden fields *again*!

**Version 2.16**

1. Fixed hidden fields *yet again*.
2. Fixed subtle problems in the file upload method that caused intermittent failures (thanks to Keven Hendrick for this one).
3. Made file upload more robust in the face of bizarre behavior by the Macintosh and Windows Netscape clients.
4. Moved the POD documentation to the bottom of the module at the request of Stephen Dahmen.
5. Added the -xbase parameter to the start\_html() method, also at the request of Stephen Dahmen.
6. Added JavaScript form buttons at Stephen's request. I'm not sure how to use this Netscape extension correctly, however, so for now the form() method is in the module as an undocumented feature. Use at your own risk!

**Version 2.15**

1. Added the **-override** parameter to all field-generating methods.
2. Documented the user\_name() and remote\_user() methods.
3. Fixed bugs that prevented empty strings from being recognized as valid textfield contents.
4. Documented the use of framesets and added a frameset example.

**Version 2.14**

This was an internal experimental version that was never released.

**Version 2.13**

1. Fixed a bug that interfered with the value "0" being entered into text fields.

**Version 2.01**

1. Added -rows and -columns to the radio and checkbox groups. No doubt this will cause much grief because it seems to promise a level of meta-organization that it doesn't actually provide.
2. Fixed a bug in the redirect() method -- it was not truly HTTP/1.0 compliant.

**Version 2.0**

The changes seemed to touch every line of code, so I decided to bump up the major version number.

1. Support for [named parameter style method calls.](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#named_param) This turns out to be a big win for extending CGI.pm when Netscape adds new HTML "features".
2. Changed behavior of hidden fields back to the correct "sticky" behavior. [This is going to break some programs,](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#hidden_fields_warning) but it is for the best in the long run.
3. Netscape 2.0b2 broke the file upload feature. CGI.pm now handles both 2.0b1 and 2.0b2-style uploading. It will probably break again in 2.0b3.
4. There were still problems with library being unable to distinguish between a form being loaded for the first time, and a subsequent loading with all fields blank. We now forcibly create a default name for the Submit button (if not provided) so that there's always at least one parameter.
5. More workarounds to prevent annoying spurious warning messages when run under the -w switch. -w is seriously broken in perl 5.001!

**Version 1.57**

1. Support for the Netscape 2.0 "File upload" field.
2. The handling of defaults for selected items in scrolling lists and multiple checkboxes is now consistent.

**Version 1.56**

1. Created true "pod" documentation for the module.
2. Cleaned up the code to avoid many of the spurious "use of uninitialized variable" warnings when running with the -w switch.
3. Added the autoEscape() method. v
4. Added string interpolation of the CGI object.
5. Added the ability to pass additional parameters to the <BODY> tag.
6. Added the ability to specify the status code in the HTTP header.

**Bug fixes in version 1.55**

1. Every time self\_url() was called, the parameter list would grow. This was a bad "feature".
2. Documented the fact that you can pass "-" to radio\_group() in order to prevent any button from being highlighted by default.

**Bug fixes in version 1.54**

1. The user\_agent() method is now documented;
2. A potential security hole in import() is now plugged.
3. Changed name of import() to import\_names() for compatability with CGI:: modules.

**Bug fixes in version 1.53**

1. Fixed several typos in the code that were causing the following subroutines to fail in some circumstances
   1. checkbox()
   2. hidden()
2. No features added

**New features added in version 1.52**

1. Added backslashing, quotation marks, and other shell-style escape sequences to the parameters passed in during debugging off-line.
2. Changed the way that the hidden() method works so that the default value always overrides the current one.
3. Improved the handling of sticky values in forms. It's now less likely that sticky values will get stuck.
4. If you call server\_name(), script\_name() and several other methods when running offline, the methods now create "dummy" values to work with.

**Bugs fixed in version 1.51**

1. param() when called without arguments was returning an array of length 1 even when there were no parameters to be had. Bad bug! Bad!
2. The HTML code generated would break if input fields contained the forbidden characters ">< or &. You can now use these characters freely.

**New features added in version 1.50**

1. import() method allows all the parameters to be imported into a namespace in one fell swoop.
2. Parameters are now returned in the same order in which they were defined.

**Bugs fixed in version 1.45**

1. delete() method didn't work correctly. This is now fixed.
2. reset() method didn't allow you to set the name of the button. Fixed.

**Bugs fixed in version 1.44**

1. self\_url() didn't include the path information. This is now fixed.

**New features added in version 1.43**

1. Added the delete() method.

**New features added in version 1.42**

1. The image\_button() method to create clickable images.
2. A few bug fixes involving forms embedded in <PRE> blocks.

**New features added in version 1.4**

1. New header shortcut methods
   * redirect() to create HTTP redirection messages.
   * start\_html() to create the HTML title, complete with the recommended <LINK> tag that no one ever remembers to include.
   * end\_html() for completeness' sake.
2. A new save() method that allows you to write out the state of an script to a file or pipe.
3. An improved version of the new() method that allows you to restore the state of a script from a file or pipe. With (2) this gives you dump and restore capabilities! (Wow, you can put a "121,931 customers served" banner at the bottom of your pages!)
4. A self\_url() method that allows you to create state-maintaining hypertext links. In addition to allowing you to maintain the state of your scripts between invocations, this lets you work around a problem that some browsers have when jumping to internal links in a document that contains a form -- the form information gets lost.
5. The user-visible labels in checkboxes, radio buttons, popup menus and scrolling lists have now been decoupled from the values sent to your CGI script. Your script can know a checkbox by the name of "cb1" while the user knows it by a more descriptive name. I've also added some parameters that were missing from the text fields, such as MAXLENGTH.
6. A whole bunch of methods have been added to get at environment variables involved in user verification and other obscure features.

**Bug fixes**

1. The problems with the hidden fields have (I hope at last) been fixed.
2. You can create multiple query objects and they will all be initialized correctly. This simplifies the creation of multiple forms on one page.
3. The URL unescaping code works correctly now.

[Table of Contents](http://pages.cs.wisc.edu/~beechung/ref/CGI.pm.html#contents)

*Lincoln D. Stein, lstein@cshl.org*[*Cold Spring Harbor Laboratory*](http://www.cshl.org/)

Last modified: Wed Jul 16 13:04:34 EDT 2003