NAME

CGI - Handle Common Gateway Interface requests and responses

SYNOPSIS

```
use strict;
use warnings;

use CGI;

my $q = CGI->new;

# Process an HTTP request
my @values = $q->multi_param('form_field');
my $value = $q->param('param_name');

my $fh = $q->upload('file_field');

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

# Prepare various HTTP responses
print $q->header();
print $q->header('application/json');

my $cookie1 = $q->cookie(
-name => 'riddle_name',
-value => "The Sphynx's Question"
);

my $cookie2 = $q->cookie(
-name => 'answers',
-value => \%answers
);

print $q->header(
-type => 'image/gif',
-expires => '+3d',
-cookie => [ $cookie1,$cookie2 ]
);

   print $q->redirect('http://somewhere.else/in/movie/land');
```

DESCRIPTION

CGI.pm is a stable, complete and mature solution for processing and preparing HTTP requests and responses. Major features including processing form submissions, file uploads, reading and writing cookies, query string generation and manipulation, and processing and preparing HTTP headers.

CGI.pm performs very well in a vanilla CGI.pm environment and also comes with built-in support for mod_perl and mod_perl2 as well as FastCGI.

It has the benefit of having developed and refined over 20 years with input from dozens of contributors and being deployed on thousands of websites. CGI.pm was included in the perl distribution from perl v5.4 to v5.20, however is has now been removed from the perl core...

CGI.pm HAS BEEN REMOVED FROM THE PERL CORE

<http://perl5.git.perl.org/perl.git/commitdiff/e9fa5a80>

If you upgrade to a new version of perl or if you rely on a system or vendor perl and get an updated version

of perl through a system update, then you will have to install CGI.pm yourself with cpan/cpanm/a vendor package/manually. To make this a little easier the CGI::Fast module has been split into its own distribution, meaning you do not need access to a compiler to install CGI.pm

The rationale for this decision is that CGI.pm is no longer considered good practice for developing web applications, **including** quick prototyping and small web scripts. There are far better, cleaner, quicker, easier, safer, more scalable, more extensible, more modern alternatives available at this point in time. These will be documented with CGI::Alternatives.

For more discussion on the removal of CGI.pm from core please see:

<http://www.nntp.perl.org/group/perl.perl5.porters/2013/05/msg202130.html>

Note that the v4 releases of CGI.pm will retain back compatibility **as much as possible**, however you may need to make some minor changes to your code if you are using deprecated methods or some of the more obscure features of the module. If you plan to upgrade to v4.00 and beyond you should read the Changes file for more information and **test your code** against CGI.pm before deploying it.

HTML Generation functions should no longer be used

All HTML generation functions within CGI.pm are no longer being maintained. Any issues, bugs, or patches will be rejected unless they relate to fundamentally broken page rendering.

The rationale for this is that the HTML generation functions of CGI.pm are an obfuscation at best and a maintenance nightmare at worst. You should be using a template engine for better separation of concerns. See CGI::Alternatives for an example of using CGI.pm with the Template::Toolkit module.

These functions, and perldoc(1) for them, will continue to exist in the v4 releases of CGI.pm but may be deprecated (soft) in v5 and beyond. All documentation for these functions has been moved to CGI::HTML::Functions.

Programming style

There are two styles of programming with CGI.pm, an object-oriented (OO) style and a function-oriented style. You are recommended to use the OO style as CGI.pm will create an internal default object when the functions are called procedurally and you will not have to worry about method names clashing with perl builtins.

In the object-oriented style you create one or more CGI objects and then use object methods to create the various elements of the page. Each CGI object starts out with the list of named parameters that were passed to your CGI script by the server. You can modify the objects, save them to a file or database and recreate them. Because each object corresponds to the ‘‘state’’ of the CGI script, and because each object’s parameter list is independent of the others, this allows you to save the state of the script and restore it later.

For example, using the object oriented style:

```
#!/usr/bin/env perl

use strict;
use warnings;

use CGI; # load CGI routines

my $q = CGI->new; # create new CGI object
print $q->header; # create the HTTP header

...
```

In the function-oriented style, there is one default CGI object that you rarely deal with directly. Instead you just call functions to retrieve CGI parameters, manage cookies, and so on. The following example is identical to above, in terms of output, but uses the function-oriented interface. The main differences are that we now need to import a set of functions into our name space (usually the ‘‘standard’’ functions), and we don’t need to create the CGI object.

```
#!/usr/bin/env perl

use strict;
use warnings;

use CGI qw/:standard/; # load standard CGI routines
print header(); # create the HTTP header

...
```

The examples in this document mainly use the object-oriented style. See HOW TO IMPORT FUNCTIONS for important information on function-oriented programming in CGI.pm

Calling CGI.pm routines

Most CGI.pm routines accept several arguments, sometimes as many as 20 optional ones! To simplify this interface, all routines use a named argument calling style that looks like this:

```
print $q->header(
-type => 'image/gif',
-expires => '+3d',
);
```

Each argument name is preceded by a dash. Neither case nor order matters in the argument list: -type, -Type, and -TYPE are all acceptable. In fact, only the first argument needs to begin with a dash. If a dash is present in the first argument CGI.pm assumes dashes for the subsequent ones.

Several routines are commonly called with just one argument. In the case of these routines you can provide the single argument without an argument name. *header()* happens to be one of these routines. In this case, the single argument is the document type.

```
print $q->header('text/html');
```

Other such routines are documented below.

Sometimes named arguments expect a scalar, sometimes a reference to an array, and sometimes a reference to a hash. Often, you can pass any type of argument and the routine will do whatever is most appropriate. For example, the *param()* routine is used to set a CGI parameter to a single or a multi-valued value. The two cases are shown below:

```
$q->param(
-name => 'veggie',
-value => 'tomato',
);

$q->param(
-name => 'veggie',
-value => [ qw/tomato tomahto potato potahto/ ],
);
```

Many routines will do something useful with a named argument that it doesn't recognize. For example, you can produce non-standard HTTP header fields by providing them as named arguments:

```
print $q->header(
-type => 'text/html',
-cost => 'Three smackers',
-annoyance_level => 'high',
-complaints_to => 'bit bucket',
);
```

This will produce the following nonstandard HTTP header:

```
HTTP/1.0 200 OK
Cost: Three smackers
Annoyance-level: high
Complaints-to: bit bucket
Content-type: text/html
```

Notice the way that underscores are translated automatically into hyphens.

Creating a new query object (object-oriented style)

```
my $query = CGI->new;
```

This will parse the input (from POST, GET and DELETE methods) and store it into a perl5(1) object called `$query`. Note that because the input parsing happens at object instantiation you have to set any CGI package variables that control parsing **before** you call CGI->new.

Any filehandles from file uploads will have their position reset to the beginning of the file.

Creating a new query object from an input file

```
my $query = CGI->new( $input_filehandle );
```

If you provide a file handle to the *new()* method, it will read parameters from the file (or STDIN, or whatever). The file can be in any of the forms describing below under debugging (i.e. a series of newline delimited TAG=VALUE pairs will work). Conveniently, this type of file is created by the *save()* method (see below). Multiple records can be saved and restored.

Perl purists will be pleased to know that this syntax accepts references to file handles, or even references to filehandle globs, which is the ''official'' way to pass a filehandle. You can also initialize the CGI object with a FileHandle or IO::File object.

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( my $in_fh,'<',"test.in") || die "Couldn't open test.in for read: $!";
restore_parameters( $in_fh );
close( $in_fh );
```

You can also initialize the query object from a hash reference:

```
my $query = CGI->new( {
'dinosaur' => 'barney',
'song' => 'I love you',
'friends' => [ qw/ Jessica George Nancy / ]
} );
```

or from a properly formatted, URL-escaped query string:

```
my $query = CGI->new('dinosaur=barney&color=purple');
```

or from a previously existing CGI object (currently this clones the parameter list, but none of the other object-specific fields, such as autoescaping):

```
my $old_query = CGI->new;
my $new_query = CGI->new($old_query);
```

To create an empty query, initialize it from an empty string or hash:

```
my $empty_query = CGI->new("");

-or-

my $empty_query = CGI->new({});
```

Fetching a list of keywords from the query

```
my @keywords = $query->keywords
```

If the script was invoked as the result of an ISINDEX search, the parsed keywords can be obtained as an array using the *keywords()* method.

Fetching the names of all the parameters passed to your script

```
my @names = $query->multi_param

my @names = $query->param
```

If the script was invoked with a parameter list (e.g. ‘‘name1=value1&name2=value2&name3=value3’’), the *param()* / *multi_param()* methods will return the parameter names as a list. 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.

The array of parameter names returned will be in the same order as they were submitted by the browser. Usually this order is the same as the order in which the parameters are defined in the form (however, this isn’t part of the spec, and so isn’t guaranteed).

Fetching the value or values of a single named parameter

```
my @values = $query->multi_param('foo');

-or-

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

Pass the *param()* / *multi_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.

Warning - calling *param()* in list context can lead to vulnerabilities if you do not sanitise user input as it is possible to inject other param keys and values into your code. This is why the *multi_param()* method exists, to make it clear that a list is being returned, note that *param()* can still be called in list context and will return a list for back compatibility.

The following code is an example of a vulnerability as the call to param will be evaluated in list context and thus possibly inject extra keys and values into the hash:

```
my %user_info = (
id => 1,
name => $query->param('name'),
);
```

The fix for the above is to force scalar context on the call to ->param by prefixing it with ‘‘scalar’’

```
name => scalar $query->param('name'),
```

If you call *param()* in list context with an argument a warning will be raised by CGI.pm, you can disable this warning by setting `$CGI::LIST_CONTEXT_WARN` to 0 or by using the *multi_param()* method instead

If a value is not given in the query string, as in the queries ‘‘name1=&name2=’’, it will be returned as an empty string.

If the parameter does not exist at all, then *param()* will return undef in 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');
```

This sets the value for the named parameter ’foo’ to an array of values. This is one way to change the value of a field AFTER the script has been invoked once before.

param() also recognizes a named parameter style of calling described in more detail later:

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

-or-

$query->param(
-name => 'foo',
-value => 'the value',
);
```

Appending additional values to a named 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. Note that this method only recognizes the named argument calling syntax.

Importing all parameters into a namespace

```
$query->import_names('R');
```

This creates a series of variables in the 'R' namespace. For example, `$R::foo`, `@R:foo`. For keyword lists, a variable `@R::keywords` will appear. If no namespace is given, this method will assume 'Q'. **WARNING**: don't import anything into 'main'; this is a major security risk!

NOTE 1: 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.

In fact, you should probably not use this method at all given the above caveats and security risks.

Deleting a parameter completely

```
$query->delete('foo','bar','baz');
```

This completely clears a list of parameters. It sometimes useful for resetting parameters that you don't want passed down between script invocations.

If you are using the function call interface, use "*Delete()*" instead to avoid conflicts with perl's built-in delete operator.

Deleting all parameters

```
$query->delete_all();
```

This clears the CGI object completely. It might be useful to ensure that all the defaults are taken when you create a fill-out form.

Use *Delete_all()* instead if you are using the function call interface.

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');
```

Likewise if PUTed data can be retrieved with code like this:

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

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

PUTDATA/POSTDATA are also available via upload_hook, and as file uploads via ‘‘-putdata_upload’’ option.

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 given in the previous sections, you can obtain a direct reference to it by calling the ***param_fetch()*** method with the name of the parameter. This will return an array reference to the named parameter, which you then can manipulate in any way you like.

You can also use a named argument style using the **-name** argument.

Fetching the parameter list as a hash

```
my $params = $q->Vars;
print $params->{'address'};
my @foo = split("\0",$params->{'foo'});
my %params = $q->Vars;

use CGI ':cgi-lib';
my $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*V ars()* 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 a list 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 list 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, and may be replaced in future versions with array references.

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).

Saving the state of the script to a file

```
$query->save(\*FILEHANDLE)
```

This will write the current state of the form to the provided filehandle. You can read it back in by providing a filehandle to the *new()* method. Note that the filehandle can be a file, a pipe, or whatever.

The format of the saved file is:

```
NAME1=VALUE1
NAME1=VALUE1'
NAME2=VALUE2
NAME3=VALUE3
=
```

Both name and value are URL escaped. Multi-valued CGI parameters are represented as repeated names. A session record is delimited by a single = symbol. You can write out multiple records and read them back in with several calls to **new**. You can do this across several sessions by opening the file in append mode, allowing you to create primitive guest books, or to keep a history of users’ queries. Here’s a short example of creating multiple session records:

```
use strict;
use warnings;
use CGI;

open (my $out_fh,'>>','test.out') || die "Can't open test.out: $!";
```

```
my $records = 5;
for ( 0 .. $records ) {
my $q = CGI->new;
$q->param( -name => 'counter',-value => $_ );
$q->save( $out_fh );
}
close( $out_fh );

# reopen for reading
open (my $in_fh,'<','test.out') || die "Can't open test.out: $!";
while (!eof($in_fh)) {
my $q = CGI->new($in_fh);
print $q->param('counter'),"\n";
}
```

The file format used for save/restore is identical to that used by the Whitehead Genome Center's data exchange format ''Boulderio'', and can be manipulated and even databased using Boulderio utilities. See Boulder for further details.

If you wish to use this method from the function-oriented (non-OO) interface, the exported name for this method is ***save_parameters()***.

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 a page, or use it as the value of the HTTP status:

```
if ( my $error = $q->cgi_error ) {
print $q->header( -status => $error );
print "Error: $error";
exit 0;
}
```

When using the function-oriented interface (see the next section), errors may only occur the first time you call *param()*. Be ready for this!

Using the function-oriented interface

To use the function-oriented interface, you must specify which CGI.pm routines or sets of routines to import into your script's namespace. There is a small overhead associated with this importation, but it isn't much.

```
use strict;
use warnings;

use CGI qw/ list of methods /;
```

The listed methods will be imported into the current package; you can call them directly without creating a CGI object first. This example shows how to import the ***param()*** and ***header()*** methods, and then use them directly:

```
use strict;
use warnings;

use CGI qw/ param header /;
print header('text/plain');
my $zipcode = param('zipcode');
```

More frequently, you'll import common sets of functions by referring to the groups by name. All function sets are preceded with a ':' character as in ':cgi' (for CGI protocol handling methods).

Here is a list of the function sets you can import:

:cgi
Import all CGI-handling methods, such as ***param()***, ***path_info()*** and the like.

:all Import all the available methods. For the full list, see the CGI.pm code, where the variable `%EXPORT_TAGS` is defined. (N.B. the :cgi-lib imports will **not** be included in the :all import, you will have to import :cgi-lib to get those)

Note that in the interests of execution speed CGI.pm does **not** use the standard Exporter syntax for specifying load symbols. This may change in the future.

Pragmas

In addition to the function sets, there are a number of pragmas that you can import. Pragmas, which are always preceded by a hyphen, change the way that CGI.pm functions in various ways. Pragmas, function sets, and individual functions can all be imported in the same *use()* line. For example, the following use statement imports the cgi set of functions and enables debugging mode (pragma -debug):

```
use strict;
use warninigs;
use CGI qw/ :cgi -debug /;
```

The current list of pragmas is as follows:

-no_undef_params
This keeps CGI.pm from including undef params in the parameter list.

-utf8
This makes CGI.pm treat all parameters as text strings rather than binary strings (see perlunitut(1) for the distinction), assuming UTF-8 for the encoding.

CGI.pm does the decoding from the UTF-8 encoded input data, restricting this decoding to input text as distinct from binary upload data which are left untouched. Therefore, a ':utf8' layer must **not** be used on STDIN.

If you do not use this option you can manually select which fields are expected to return utf-8 strings and convert them using code like this:

```
use strict;
use warnings;

use CGI;
use Encode qw/ decode /;

my $cgi = CGI->new;
my $param = $cgi->param('foo');
$param = decode( 'UTF-8',$param );
```

-putdata_upload
Makes `$cgi->param('PUTDATA');` and `$cgi->param('POSTDATA');` act like file uploads named PUTDATA and POSTDATA. See ''Handling non-urlencoded arguments'' and ''Processing a file upload field'' PUTDATA/POSTDATA are also available via upload_hook.

-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 below.

-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
```

Semicolon-delimited query strings are always accepted, and will be emitted by *self_url()* and

query_string(). newstyle_urls became the default in version 2.64.

-oldstyle_urls
Separate the name=value pairs in CGI parameter query strings with ampersands rather than semicolons. This is no longer the default.

-no_debug
This turns off the command-line processing features. If you want to run a CGI.pm script from the command line, and you don't want it to read CGI parameters from the command line or STDIN, 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 the section on debugging for more details.

GENERATING DYNAMIC DOCUMENTS

Most of CGI.pm's functions deal with creating documents on the fly. Generally you will produce the HTTP header first, followed by the document itself. CGI.pm provides functions for generating HTTP headers of various types.

Each of these functions produces a fragment of HTTP which you can print out directly so that it is processed by the browser, appended to a string, or saved to a file for later use.

Creating a standard http header

Normally the first thing you will do in any CGI script is print out an HTTP header. This tells the browser what type of document to expect, and gives other optional information, such as the language, expiration date, and whether to cache the document. The header can also be manipulated for special purposes, such as server push and pay per view pages.

```
use strict;
use warnings;

use CGI;

my $cgi = CGI->new;

print $cgi->header;

-or-

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

-or-

print $cgi->header('text/html','204 No response');

-or-

print $cgi->header(
-type => 'image/gif',
-nph => 1,
-status => '402 Payment required',
-expires => '+3d',
-cookie => $cookie,
-charset => 'utf-8',
```

```
-attachment => 'foo.gif',
-Cost => '$2.00'
);
```

header() returns the Content-type: header. You can provide your own MIME type if you choose, otherwise it defaults to text/html. An optional second parameter specifies the status code and a human-readable message. For example, you can specify 204, ‘‘No response’’ to create a script that tells the browser to do nothing at all. Note that RFC 2616 expects the human-readable phase to be there as well as the numeric status code.

The last example shows the named argument style for passing arguments to the CGI methods using named parameters. Recognized parameters are **-type**, **-status**, **-expires**, and **-cookie**. Any other named parameters will be stripped of their initial hyphens and turned into header fields, allowing you to specify any HTTP header you desire. Internal underscores will be turned into hyphens:

```
print $cgi->header( -Content_length => 3002 );
```

Most browsers will not cache the output from CGI scripts. Every time the browser reloads the page, the script is invoked anew. You can change this behavior with the **-expires** parameter. When you specify an absolute or relative expiration interval with this parameter, some 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
Thursday, 25-Apr-2018 00:40:33 GMT at the indicated time & date
```

The **-cookie** parameter generates a header that tells the browser to provide a ‘‘magic cookie’’ during all subsequent transactions with your script. Some cookies have a special format that includes interesting attributes such as expiration time. Use the *cookie()* method to create and retrieve session cookies.

The **-nph** parameter, if set to a true value, will issue the correct headers to work with a NPH (no-parse-header) script. This is important to use with certain servers that expect all their scripts to be NPH.

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 sets the *charset()* method as well. **Note** that the default being ISO-8859-1 may not make sense for all content types, e.g.:

```
Content-Type: image/gif; charset=ISO-8859-1
```

In the above case you need to pass -charset => ’’ to prevent the default being used.

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’’.

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 $cgi->header( -p3p => [ qw/ CAO DSP LAW CURa / ] );
print $cgi->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"
```

CGI.pm will accept valid multi-line headers when each line is separated with a CRLF value (‘‘\r\n’’ on most platforms) followed by at least one space. For example:

```
print $cgi->header( -ingredients => "ham\r\n\seggs\r\n\sbacon" );
```

Invalid multi-line header input will trigger in an exception. When multi-line headers are received, CGI.pm will always output them back as a single line, according to the folding rules of RFC 2616: the newlines will be removed, while the white space remains.

Generating a redirection header

```
    print $q->redirect( 'http://somewhere.else/in/movie/land'
);
```

Sometimes you don't want to produce a document yourself, but simply redirect the browser elsewhere, perhaps choosing a URL based on the time of day or the identity of the user.

The *redirect()* method redirects the browser to a different URL. If you use redirection like this, you should **not** print out a header as well.

You should always use full URLs (including the http: or ftp: part) in redirection requests. Relative URLs will not work correctly.

You can also use named arguments:

```
print $q->redirect(
       -uri    => 'http://somewhere.else/in/movie/land',
-nph => 1,
-status => '301 Moved Permanently'
);
```

All names arguments recognized by *header()* are also recognized by *redirect()*. However, most HTTP headers, including those generated by -cookie and -target, are ignored by the browser.

The **-nph** parameter, if set to a true value, will issue the correct headers to work with a NPH (no-parse-header) script. This is important to use with certain servers, such as Microsoft IIS, which expect all their scripts to be NPH.

The **-status** parameter will set the status of the redirect. HTTP defines several different possible redirection status codes, and the default if not specified is 302, which means ‘‘moved temporarily.’’ You may change the status to another status code if you wish.

Note that the human-readable phrase is also expected to be present to conform with RFC 2616, section 6.1.

Creating a self-referencing url that preserves state information

```
my $myself = $q->self_url;
print qq(<a href="$myself">I'm talking to myself.</a>);
```

self_url() will return a URL, that, when selected, will re-invoke this script with all its state information intact. This is most useful when you want to jump around within the document using internal anchors but you don't want to disrupt the current contents of the form(s). Something like this will do the trick:

```
my $myself = $q->self_url;
print "<a href=\"$myself#table1\">See table 1</a>";
print "<a href=\"$myself#table2\">See table 2</a>";
print "<a href=\"$myself#yourself\">See for yourself</a>";
```

If you want more control over what's returned, using the ***url()*** method instead.

You can also retrieve a query string representation of the current object state with *query_string()*:

```
my $the_string = $q->query_string();
```

The behavior of calling query_string is currently undefined when the HTTP method is something other than GET.

If you want to retrieved the query string as set in the webserver, namely the environment variable, you can call *env_query_string()*

Obtaining the script's url

```
my $full_url = url();
my $full_url = url( -full =>1 ); # alternative syntax
my $relative_url = url( -relative => 1 );
my $absolute_url = url( -absolute =>1 );
my $url_with_path = url( -path_info => 1 );
my $url_path_qry = url( -path_info => 1, -query =>1 );
my $netloc = url( -base => 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 re-invoke 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.

-base

Generate just the protocol and net location, as in http://www.foo.com:8000

-rewrite

If Apache's mod_rewrite is turned on, then the script name and path info probably won't match the request that the user sent. Set -rewrite => 1 (default) to return URLs that match what the user sent (the original request URI). Set -rewrite => 0 to return URLs that match the URL after the mod_rewrite rules have run.

Mixing post and url parameters

```
my $color = 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.

Processing a file upload field

Basics

When the form is processed, you can retrieve an IO::File compatible handle for a file upload field like this:

```
use autodie;

# undef may be returned if it's not a valid file handle
if ( my $io_handle = $q->upload('field_name') ) {
open ( my $out_file,'>>','/usr/local/web/users/feedback' );
while ( my $bytesread = $io_handle->read($buffer,1024) ) {
print $out_file $buffer;
}
}
```

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

If you want the entered file name for the file, you can just call *param()*:

```
my $filename = $q->param('field_name');
```

Different browsers will return slightly different things for the name. Some browsers return the filename only. Others return the full path to the file, using the path conventions of the user's machine. Regardless, the name returned is always the name of the file on the *user's* machine, and is unrelated to the name of the temporary file that CGI.pm creates during upload spooling (see below).

When a file is uploaded the browser usually sends along some information along with it in the format of headers. The information usually includes the MIME content type. To retrieve this information, call *uploadInfo()*. It returns a reference to a hash containing all the document headers.

```
my $filename = $q->param( 'uploaded_file' );
my $type = $q->uploadInfo( $filename )->{'Content-Type'};
if ( $type ne 'text/html' ) {
die "HTML FILES ONLY!";
}
```

Note that you must use ->param to get the filename to pass into uploadInfo as internally this is represented as a File::Temp object (which is what will be returned by ->param). When using ->Vars you will get the literal filename rather than the File::Temp object, which will not return anything when passed to uploadInfo. So don't use ->Vars.

If you are using a machine that recognizes ''text'' and ''binary'' data modes, be sure to understand when and how to use them (see the Camel book). Otherwise you may find that binary files are corrupted during file uploads.

Accessing the temp files directly

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. You can access the temp file for a file upload by passing the file name to the *tmpFileName()* method:

```
my $filename = $query->param( 'uploaded_file' );
my $tmpfilename = $query->tmpFileName( $filename );
```

The temporary file will be deleted automatically when your program exits unless you manually rename it or set `$CGI::UNLINK_TMP_FILES` to 0. 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.

Changes in temporary file handling (v4.05+)

CGI.pm had its temporary file handling significantly refactored, this logic is now all deferred to File::Temp (which is wrapped in a compatibility object, CGI::File::Temp - **DO NOT USE THIS PACKAGE DIRECTLY**). As a consequence the PRIVATE_TEMPFILES variable has been removed along with

deprecation of the private_tempfiles routine and **complete** removal of the CGITempFile package. The `$CGITempFile::TMPDIRECTORY` is no longer used to set the temp directory, refer to the perldoc(1) for File::Temp is you want to override the default settings in that package (the TMPDIR env variable is still available on some platforms). For Windows platforms the temporary directory order remains as before: TEMP > TMP > WINDIR (> TMPDIR) so if you have any of these in use in existing scripts they should still work.

The Fh package still exists but does nothing, the CGI::File::Temp class is a subclass of both File::Temp and the empty Fh package, so if you have any code that checks that the filehandle isa Fh this should still work.

When you get the internal file handle you will receive a File::Temp object, this should be transparent as File::Temp isa IO::Handle and isa IO::Seekable meaning it behaves as previously. If you are doing anything out of the ordinary with regards to temp files you should test your code before deploying this update and refer to the File::Temp documentation for more information.

Handling interrupted file uploads

There are occasionally problems involving parsing the uploaded file. This usually happens when the user presses ‘‘Stop’’ before the upload is finished. In this case, CGI.pm will return undef for the name of the uploaded file and set *cgi_error()* to the string ‘‘400 Bad request (malformed multipart POST)’’. This error message is designed so that you can incorporate it into a status code to be sent to the browser. Example:

```
my $file = $q->upload( 'uploaded_file' );
if ( !$file && $q->cgi_error ) {
print $q->header( -status => $q->cgi_error );
exit 0;
}
```

Progress bars for file uploads and avoiding temp files

CGI.pm gives you low-level access to file upload management through a file upload hook. You can use this feature to completely turn off the temp file storage of file uploads, or potentially write your own file upload progress meter.

This is much like the UPLOAD_HOOK facility available in Apache::Request, with the exception that the first argument to the callback is an Apache::Upload object, here it’s the remote filename.

```
my $q = CGI->new( \&hook [,$data [,$use_tempfile]] );

sub hook {
my ( $filename, $buffer, $bytes_read, $data ) = @_;
print "Read $bytes_read bytes of $filename\n";
}
```

The `$data` field is optional; it lets you pass configuration information (e.g. a database handle) to your hook callback.

The `$use_tempfile` field is a flag that lets you turn on and off CGI.pm’s use of a temporary disk-based file during file upload. If you set this to a FALSE value (default true) then `$q`->param(’uploaded_file’) will no longer work, and the only way to get at the uploaded data is via the hook you provide.

If using the function-oriented interface, call the *CGI::upload_hook()* method before calling *param()* or any other CGI functions:

```
CGI::upload_hook( \&hook [,$data [,$use_tempfile]] );
```

This method is not exported by default. You will have to import it explicitly if you wish to use it without the CGI:: prefix.

Troubleshooting file uploads on Windows

If you are using CGI.pm on a Windows platform and find that binary files get slightly larger when uploaded but that text files remain the same, then you have forgotten to activate binary mode on the output filehandle. Be sure to call *binmode()* on any handle that you create to write the uploaded file to disk.

Older ways to process file uploads

This section is here for completeness. if you are building a new application with CGI.pm, you can skip it.

The original way to process file uploads with CGI.pm was to use *param()*. The value it returns has a dual nature as both a file name and a lightweight filehandle. This dual nature is problematic if you following the recommended practice of having `use strict` in your code. perl will complain when you try to use a string as a filehandle. 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 solve this problem the *upload()* method was added, which always returns a lightweight filehandle. This generally works well, but will have trouble interoperating with some other modules because the file handle is not derived from IO::File. So that brings us to current recommendation given above, which is to call the *handle()* method on the file handle returned by *upload()*. That upgrades the handle to an IO::File. It's a big win for compatibility for a small penalty of loading IO::File the first time you call it.

HTTP COOKIES

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:

1. 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.

2. 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.

3. 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.

4. 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:

```
my $cookie = $q->cookie(
-name => 'sessionID',
-value => 'xyzzy',
-expires => '+1h',
-path => '/cgi-bin/database',
-domain => '.capricorn.org',
-secure => 1
);

print $q->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 browsers limit their 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 hash reference. For example, you can store an entire hash into a cookie this way:

```
my $cookie = $q->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()* method:

```
use strict;
use warnings;

use CGI;

my $q = CGI->new;
my $cookie = ...
print $q->header( -cookie => $cookie );
```

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

```
my $cookie1 = $q->cookie(
-name => 'riddle_name',
-value => "The Sphynx's Question"
);

my $cookie2 = $q->cookie(
```

```
-name => 'answers',
-value => \%answers
);

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

To retrieve a cookie, request it by name by calling *cookie()* method without the **-value** parameter. This example uses the object-oriented form:

```
my $riddle = $q->cookie('riddle_name');
my %answers = $query->cookie('answers');
```

Cookies created with a single scalar value, such as the ‘‘riddle_name’’ cookie, will be returned in that form. Cookies with array and hash values can also be retrieved.

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
my $c = cookie( -name => 'answers',-value => [$q->param('answers')] );
# vice-versa
$q->param( -name => 'answers',-value => [ $q->cookie('answers')] );
```

If you call *cookie()* without any parameters, it will return a list of the names of all cookies passed to your script:

```
my @cookies = $q->cookie();
```

See the **cookie.cgi** example script for some ideas on how to use cookies effectively.

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:

```
your_script.pl keyword1 keyword2 keyword3
```

or this:

```
your_script.pl keyword1+keyword2+keyword3
```

or this:

```
your_script.pl name1=value1 name2=value2
```

or this:

```
your_script.pl name1=value1&name2=value2
```

To turn off this feature, use the -no_debug pragma.

To test the POST method, you may enable full debugging with the -debug pragma. This will allow you to feed newline-delimited name=value pairs to the script on standard input.

When debugging, you can use quotes and backslashes to escape characters in the familiar shell manner, letting you place spaces and other funny characters in your parameter=value pairs:

```
your_script.pl "name1='I am a long value'" "name2=two\ words"
```

Finally, you can set the path info for the script by prefixing the first name/value parameter with the path followed by a question mark (?):

```
your_script.pl /your/path/here?name1=value1&name2=value2
```

FETCHING ENVIRONMENT 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 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 that the capitalization changed between version 2.43 and 2.44 in order to avoid conflict with perl's *accept()* function.

raw_cookie()
Returns the HTTP_COOKIE variable. Cookies have a special format, and this method call just returns the raw form (?cookie dough). See *cookie()* for ways of setting and retrieving cooked cookies.

Called with no parameters, *raw_cookie()* returns the packed cookie structure. You can separate it into individual cookies by splitting on the character sequence "; ". Called with the name of a cookie, retrieves the **unescaped** form of the cookie. You can use the regular *cookie()* method to get the names, or use the *raw_fetch()* method from the CGI::Cookie module.

env_query_string()
Returns the QUERY_STRING variable, note that this is the original value as set in the environment by the webserver and (possibly) not the same value as returned by *query_string()*, which represents the object state

user_agent()
Returns the HTTP_USER_AGENT variable. If you give this method a single argument, it will attempt to pattern match on it, allowing you to do something like user_agent(Mozilla);

path_info()
Returns additional path information from the script URL. E.G. fetching /cgi-bin/your_script/additional/stuff will result in *path_info()* returning "/additional/stuff".

NOTE: The Microsoft Internet Information Server is broken with respect to additional path information. If you use the perl DLL library, the IIS server will attempt to execute the additional path information as a perl script. If you use the ordinary file associations mapping, the path information will be present in the environment, but incorrect. The best thing to do is to avoid using additional path information in CGI scripts destined for use with IIS. A best attempt has been made to make CGI.pm do the right thing.

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".

The Microsoft IIS is broken with respect to the translated path as well.

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

remote_ident()
Returns the name of the remote user (as returned by identd) or undef if not set

remote_addr()
Returns the remote host IP address, or 127.0.0.1 if the address is unavailable.

request_uri()
Returns the interpreted pathname of the requested document or CGI (relative to the document root). Or undef if not set.

script_name()
Return the script name as a partial URL, for self-referring scripts.

referer()
Return the URL of the page the browser was viewing prior to fetching your script.

auth_type()
Return the authorization/verification method in use for this script, if any.

server_name()
Returns the name of the server, usually the machine's host name.

virtual_host()
When using virtual hosts, returns the name of the host that the browser attempted to contact

server_port()
Return the port that the server is listening on.

server_protocol()
Returns the protocol and revision of the incoming request, or defaults to HTTP/1.0 if this is not set

virtual_port()
Like *server_port()* except that it takes virtual hosts into account. Use this when running with virtual hosts.

server_software()
Returns the server software and version number.

remote_user()
Return the authorization/verification name used for user verification, if this script is protected.

user_name()
Attempt to obtain the remote user's name, using a variety of different techniques. May not work in all browsers.

request_method()
Returns the method used to access your script, usually one of 'POST', 'GET' or 'HEAD'.

content_type()
Returns the content_type of data submitted in a POST, generally multipart/form-data or application/x-www-form-urlencoded

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:

```
my $requested_language = $q->http('Accept-language');

my $requested_language = $q->http('Accept_language');

my $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.

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. Many Unix servers look at the beginning of the script's name for the prefix ''nph-''. The Macintosh WebSTAR server and Microsoft's Internet Information Server, in contrast, try to decide whether a program is an NPH script by examining the first line of script output.

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.

The Microsoft Internet Information Server requires NPH mode. As of version 2.30, CGI.pm will automatically detect when the script is running under IIS and put itself into this mode. You do not need to do this manually, although it won't hurt anything if you do.

In the **use** statement

Simply add the ''-nph'' pragma 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 header(-nph=>1);
```

SERVER PUSH

CGI.pm provides four simple functions for producing multipart documents of the type needed to implement server push. These functions were graciously provided by Ed Jordan <ed@fidalgo.net>. 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/bin/env perl

use strict;
use warnings;

use CGI qw/:push -nph/;

$| = 1;
print multipart_init( -boundary=>'----here we go!' );
for (0 .. 4) {
print multipart_start( -type=>'text/plain' ),
"The current time is ",scalar( localtime ),"\n";
if ($_ < 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, -charset => $charset );
```

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.

The -charset provides the character set, if not provided this will default to ISO-8859-1

multipart_start()

```
multipart_start( -type => $type, -charset => $charset );
```

Start a new part of the multipart document using the specified MIME type and charset. If not specified, text/html ISO-8859-1 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.

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 gigabytes. 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 10 megabytes.

$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.

To use these variables, set the variable at the top of the script, right after the ''use'' statement:

```
#!/usr/bin/env perl

use strict;
use warnings;

use CGI;

$CGI::POST_MAX = 1024 * 1024 * 10; # max 10MB posts
$CGI::DISABLE_UPLOADS = 1; # no uploads
```

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:

```
my $uploaded_file = $q->param('upload');
if ( !$uploaded_file && $q->cgi_error() ) {
print $q->header( -status => $q->cgi_error() );
exit 0;
}
```

However it isn’t clear that any browser currently knows what to do with this status code. It might be better just to create a page that warns the user of the problem.

COMPATIBILITY WITH CGI−LIB.PL

To make it easier to port existing programs that use cgi-lib.pl the compatibility routine ‘‘ReadParse’’ is provided. Porting is simple:

OLD VERSION

```
require "cgi-lib.pl";
&ReadParse;
print "The value of the antique is $in{antique}.\n";
```

NEW VERSION

```
use CGI;
CGI::ReadParse();
print "The value of the antique is $in{antique}.\n";
```

CGI.pm’s *ReadParse()* routine creates a tied variable named `%in`, which can be accessed to obtain the query variables. Like ReadParse, you can also provide your own variable. Infrequently used features of ReadParse, such as the creation of `@in` and `$in` variables, are not supported.

Once you use ReadParse, you can retrieve the query object itself this way:

```
my $q = $in{CGI};
```

This allows you to start using the more interesting features of CGI.pm without rewriting your old scripts from scratch.

An even simpler way to mix cgi-lib calls with CGI.pm calls is to import both the `:cgi-lib` and `:standard` method:

```
use CGI qw(:cgi-lib :standard);
&ReadParse;
print "The price of your purchase is $in{price}.\n";
print textfield(-name=>'price', -default=>'$1.99');
```

Cgi-lib functions that are available in CGI.pm

In compatibility mode, the following cgi-lib.pl functions are available for your use:

```
ReadParse()
PrintHeader()
SplitParam()
MethGet()
MethPost()
```

LICENSE

CREDITS

Thanks very much to:

Mark Stosberg (mark@stosberg.com)
Matt Heffron (heffron@falstaff.css.beckman.com)
James Taylor (james.taylor@srs.gov)
Scott Anguish <sanguish@digifix.com>
Mike Jewell (mlj3u@virginia.edu)
Timothy Shimmin (tes@kbs.citri.edu.au)
Joergen Haegg (jh@axis.se)
Laurent Delfosse (delfosse@delfosse.com)
Richard Resnick (applepi1@aol.com)
Craig Bishop (csb@barwonwater.vic.gov.au)
Tony Curtis (tc@vcpc.univie.ac.at)
Tim Bunce (Tim.Bunce@ig.co.uk)
Tom Christiansen (tchrist@convex.com)
Andreas Koenig (k@franz.ww.TU-Berlin.DE)
Tim MacKenzie (Tim.MacKenzie@fulcrum.com.au)
Kevin B. Hendricks (kbhend@dogwood.tyler.wm.edu)
Stephen Dahmen (joyfire@inxpress.net)
Ed Jordan (ed@fidalgo.net)
David Alan Pisoni (david@cnation.com)
Doug MacEachern (dougm@opengroup.org)
Robin Houston (robin@oneworld.org)
...and many many more...
for suggestions and bug fixes.

BUGS

Address bug reports and comments to: <https://github.com/leejo/CGI.pm/issues>

The original bug tracker can be found at: <https://rt.cpan.org/Public/Dist/Display.html?Queue=CGI.pm>

When sending bug reports, please provide the version of CGI.pm, the version of perl, the name and version of your Web server, and the name and version of the operating system you are using. If the problem is even remotely browser dependent, please provide information about the affected browsers as well.

Failing tests cases are appreciated with issues, and if you submit a patch then it will *not* be accepted unless you provide a reasonable automated test case with it (please see the existing tests in t/ for examples)

SEE ALSO

CGI::Carp - provides Carp implementation tailored to the CGI environment.

CGI::Fast - supports running CGI applications under FastCGI