NAME

CGI - Handle Common Gateway Interface requests and responses

SYNOPSIS

```
use CGI;

my $q = CGI->new;

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

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

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

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

$cookie1 = $q->cookie(-name=>'riddle_name', -value=>"The Sphynx's Question");
$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. Some HTML generation utilities are included as well.

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 10 years with input from dozens of contributors and being deployed on thousands of websites. CGI.pm has been included in the Perl distribution since Perl 5.4, and has become a de-facto standard.

PROGRAMMING STYLE

There are two styles of programming with CGI.pm, an object-oriented style and a function-oriented style. 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, here is how you create a simple "Hello World" HTML page:

```
#!/usr/local/bin/perl -w
use CGI; # load CGI routines
$q = CGI->new; # create new CGI object
print $q->header, # create the HTTP header
$q->start_html('hello world'), # start the HTML
$q->h1('hello world'), # level 1 header
$q->end_html; # end the HTML
```

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, create HTML tags, manage cookies, and so on. This provides you with a cleaner programming interface, but limits you to using one CGI object at a time. The following example prints the same page, 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/local/bin/perl
use CGI qw/:standard/; # load standard CGI routines
print header, # create the HTTP header
start_html('hello world'), # start the HTML
h1('hello world'), # level 1 header
end_html; # end the HTML
```

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=>['tomato','tomahto','potato','potahto']);
```

A large number of routines in CGI.pm actually aren't specifically defined in the module, but are generated automatically as needed. These are the "HTML shortcuts," routines that generate HTML tags for use in dynamically-generated pages. HTML tags have both attributes (the attribute="value" pairs within the tag itself) and contents (the part between the opening and closing pairs.) To distinguish between attributes and contents, CGI.pm uses the convention of passing HTML attributes as a hash reference as the first argument, and the contents, if any, as any subsequent arguments. It works out like this:

```
Code Generated HTML
---- --------------
h1() <h1>
h1('some','contents'); <h1>some contents</h1>
h1({-align=>left}); <h1 align="LEFT">
h1({-align=>left},'contents'); <h1 align="LEFT">contents</h1>
```

HTML tags are described in more detail later.

Many newcomers to CGI.pm are puzzled by the difference between the calling conventions for the HTML shortcuts, which require curly braces around the HTML tag attributes, and the calling conventions for other routines, which manage to generate attributes without the curly brackets. Don't be confused. As a convenience the curly braces are optional in all but the HTML shortcuts. If you like, you can use curly braces when calling any routine that takes named arguments. For example:

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

If you use the **-w** switch, you will be warned that some CGI.pm argument names conflict with built-in Perl functions. The most frequent of these is the -values argument, used to create multi-valued menus, radio button clusters and the like. To get around this warning, you have several choices:

1. Use another name for the argument, if one is available. For example, -value is an alias for -values.
2. Change the capitalization, e.g. -Values
3. Put quotes around the argument name, e.g. '-values'

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. HTML-generating routines perform a different type of translation.

This feature allows you to keep up with the rapidly changing HTTP and HTML "standards".

CREATING A NEW QUERY OBJECT (OBJECT-ORIENTED STYLE):

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

This will parse the input (from POST, GET and DELETE methods) and store it into a perl5(1) object called `$query`.

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

```
$query = CGI->new(INPUTFILE);
```

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:

```
$query = CGI->new(\*STDIN);
```

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 (IN,"test.in") || die;
restore_parameters(IN);
close IN;
```

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

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

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

```
$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):

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

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

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

-or-

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

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 as an array using the *keywords()* method.

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. 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.

NOTE: As of version 1.5, 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:

```
@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=", it will be returned as an empty string.

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');
```

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. (Another way is with the -override parameter accepted by all methods that generate form elements.)

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.

NOTE 2: In older versions, this method was called ***import()***. As of version 2.20, this name has been removed completely to avoid conflict with the built-in Perl module **import** operator.

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, don't worry about it. It only affects people trying to use CGI for XML processing and other specialized tasks.)

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:

```
$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 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.

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 CGI;

open (OUT,'>>','test.out') || die;
$records = 5;
for (0..$records) {
my $q = CGI->new;
$q->param(-name=>'counter',-value=>$_);
$q->save(\*OUT);
}
close OUT;

# reopen for reading
open (IN,'<','test.out') || die;
while (!eof(IN)) {
my $q = CGI->new(\*IN);
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

http://stein.cshl.org/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 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 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 CGI <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 CGI 'param','header';
print header('text/plain');
$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 ":html3" (for tags defined in the HTML 3 standard).

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.

:form
: Import all fill-out form generating methods, such as ***textfield()***.

:html2
: Import all methods that generate HTML 2.0 standard elements.

:html3
: Import all methods that generate HTML 3.0 elements (such as <table>, <super> and <sub>).

:html4
: Import all methods that generate HTML 4 elements (such as <abbrev>, <acronym> and <thead>).

:netscape
: Import the <blink>, <fontsize> and tags.

:html
: Import all HTML-generating shortcuts (i.e. 'html2', 'html3', 'html4' and 'netscape')

:standard
: Import "standard" features, 'html2', 'html3', 'html4', 'form' and 'cgi'.

:all
: Import all the available methods. For the full list, see the CGI.pm code, where the variable `%EXPORT_TAGS` is defined.

If you import a function name that is not part of CGI.pm, the module will treat it as a new HTML tag and generate the appropriate subroutine. You can then use it like any other HTML tag. This is to provide for the rapidly-evolving HTML "standard." For example, say Microsoft comes out with a new tag called <gradient> (which causes the user's desktop to be flooded with a rotating gradient fill until his machine reboots). You don't need to wait for a new version of CGI.pm to start using it immediately:

```
use CGI qw/:standard :html3 gradient/;
print gradient({-start=>'red',-end=>'blue'});
```

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.

If you import any of the state-maintaining CGI or form-generating methods, a default CGI object will be created and initialized automatically the first time you use any of the methods that

require one to be present. This includes ***param()***, ***textfield()***, ***submit()*** and the like. (If you need direct access to the CGI object, you can find it in the global variable **$CGI::Q**). By importing CGI.pm methods, you can create visually elegant scripts:

```
use CGI qw/:standard/;
print
header,
start_html('Simple Script'),
h1('Simple Script'),
start_form,
"What's your name? ",textfield('name'),p,
"What's the combination?",
checkbox_group(-name=>'words',
-values=>['eenie','meenie','minie','moe'],
-defaults=>['eenie','moe']),p,
"What's your favorite color?",
popup_menu(-name=>'color',
-values=>['red','green','blue','chartreuse']),p,
submit,
end_form,
hr,"\n";

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')),".\n";
}
print end_html;
```

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 standard set of functions and enables debugging mode (pragma -debug):

```
use CGI qw/:standard -debug/;
```

The current list of pragmas is as follows:

-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* HTML extension. This lets you go wild with new and unsupported tags:

```
use CGI qw(-any);
$q=CGI->new;
print $q->gradient({speed=>'fast',start=>'red',end=>'blue'});
```

Since using <cite>any</cite> 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 Malcolm Beattie's Perl compiler. Use it in conjunction with the methods or method families 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()* method instead:

```
use CGI();
CGI->compile();
```

This is particularly useful in a mod_perl environment, in which you might want to precompile all CGI routines in a startup script, and then import the functions individually in each mod_perl script.

-nosticky

By default the CGI module implements a state-preserving behavior called "sticky" fields. The way this works is that if you are regenerating a form, the methods that generate the form field values will interrogate *param()* to see if similarly-named parameters are present in the query string. If they find a like-named parameter, they will use it to set their default values.

Sometimes this isn't what you want. The **-nosticky** pragma prevents this behavior. You can also selectively change the sticky behavior in each element that you generate.

-tabindex

Automatically add tab index attributes to each form field. With this option turned off, you can still add tab indexes manually by passing a -tabindex option to each field-generating method.

-no_undef_params

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

-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.

If *start_html()*'s -dtd parameter specifies an HTML 2.0, 3.2, 4.0 or 4.01 DTD, XHTML will automatically be disabled without needing to use this pragma.

-utf8

This makes CGI.pm treat all parameters as UTF-8 strings. Use this with care, as it will interfere with the processing of binary uploads. It is better to manually select which fields are expected to return utf-8 strings and convert them using code like this:

```
use Encode;
my $arg = decode utf8=>param('foo');
```

-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.

-autoload
: This 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.

-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 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.

-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. 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, reducing, but not eliminating the risk of eavesdropping (there is still a potential race condition). To make life harder for the attacker, the program chooses tempfile names by calculating a 32 bit checksum of the incoming HTTP headers.

  To ensure that the temporary file cannot be read by other CGI scripts, use suEXEC or a CGI wrapper program to run your script. The temporary file is created with mode 0600 (neither world nor group readable).

  The temporary directory is selected using the following algorithm:

```
1. if $CGITempFile::TMPDIRECTORY is already set, use that

2. if the environment variable TMPDIR exists, use the location
indicated.

3. Otherwise try the locations /usr/tmp, /var/tmp, C:\temp,
/tmp, /temp, ::Temporary Items, and \WWW_ROOT.
```

  Each of these locations is checked that it is a directory and is writable. If not, the algorithm tries the next choice.

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_*tag_name* and end_*tag_name*, as in:

```
print start_h1,'Level 1 Header',end_h1;
```

With a few exceptions (described below), start_*tag_name* and end_*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 tag)

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 as well as for generating HTML. For creating GIF images, see the GD.pm module.

Each of these functions produces a fragment of HTML or HTTP which you can print out directly so that it displays in the browser window, append to a string, or save 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.

```
print header;

-or-

print header('image/gif');

-or-

print header('text/html','204 No response');

-or-

print header(-type=>'image/gif',
-nph=>1,
-status=>'402 Payment required',
-expires=>'+3d',
-cookie=>$cookie,
-charset=>'utf-7',
-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 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-1999 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 w ell.

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 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"
```

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

```
print 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 three different possible redirection status codes:

```
301 Moved Permanently
302 Found
303 See Other
```

The default if not specified is 302, which means "moved temporarily." You may change the status to another status code if you wish. Be advised that changing the status to anything other than 301, 302 or 303 will probably break redirection.

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

CREATING THE HTML DOCUMENT HEADER

```
print start_html(-title=>'Secrets of the Pyramids',
-author=>'fred@capricorn.org',
-base=>'true',
-target=>'_blank',
-meta=>{'keywords'=>'pharaoh secret mummy',
'copyright'=>'copyright 1996 King Tut'},
-style=>{'src'=>'/styles/style1.css'},
-BGCOLOR=>'blue');
```

The *start_html()* routine creates the top of the page, along with a lot of optional information that controls the page's appearance and behavior.

This method returns a canned HTML header and the opening <body> tag. All parameters are optional. In the named parameter form, recognized parameters are -title, -author, -base, -xbase, -dtd, -lang and -target (see below for the explanation). Any additional parameters you provide, such as the unofficial BGCOLOR attribute, are added to the <body> tag. Additional parameters must be proceeded by a hyphen.

The argument **-xbase** allows you to provide an HREF for the <base> tag different from the current location, as in

```
-xbase=>"" -- http://home.mcom.com/ -P
```

All relative links will be interpreted relative to this tag.

The argument **-target** allows you to provide a default target frame for all the links and fill-out forms on the page. **This is a non-standard HTTP feature which only works with some browsers!**

```
-target=>"answer_window"
```

All relative links will be interpreted relative to this tag. You add arbitrary meta information to the header with the **-meta** argument. This argument expects a reference to a hash containing name/value pairs of meta information. These will be turned into a series of header <meta> tags that look something like this:

```
<meta name="keywords" content="pharaoh secret mummy">
<meta name="description" content="copyright 1996 King Tut">
```

To create an HTTP-EQUIV type of <meta> tag, use **-head**, described below.

The **-style** argument is used to incorporate cascading stylesheets into your code. See the section on CASCADING STYLESHEETS for more information.

The **-lang** argument is used to incorporate a language attribute into the <html> tag. For example:

```
print $q->start_html(-lang=>'fr-CA');
```

The default if not specified is "en-US" for US English, unless the -dtd parameter specifies an HTML 2.0 or 3.2 DTD, in which case the lang attribute is left off. You can force the lang attribute to left off in other cases by passing an empty string (-lang=>'').

The **-encoding** argument can be used to specify the character set for XHTML. It defaults to iso-8859-1 if not specified.

The **-dtd** argument can be used to specify a public DTD identifier string. For example:

```
-dtd => '-//W3C//DTD HTML 4.01 Transitional//EN')
```

Alternatively, it can take public and system DTD identifiers as an array:

```
    dtd => [ '-//W3C//DTD HTML 4.01 Transitional//EN', 'http://www.w3.org/TR/html4/loose.dtd'
]
```

For the public DTD identifier to be considered, it must be valid. Otherwise it will be replaced by the default DTD. If the public DTD contains 'XHTML', CGI.pm will emit XML.

The **-declare_xml** argument, when used in conjunction with XHTML, will put a <?xml> declaration at the top of the HTML header. The sole purpose of this declaration is to declare the character set encoding. In the absence of -declare_xml, the output HTML will contain a <meta> tag that specifies the encoding, allowing the HTML to pass most validators. The default for -declare_xml is false.

You can place other arbitrary HTML elements to the <head> section with the **-head** tag. For example, to place a <link> element in the head section, use this:

```
print start_html(-head=>Link({-rel=>'shortcut icon',
-href=>'favicon.ico'}));
```

To incorporate multiple HTML elements into the <head> section, just pass an array reference:

```
print start_html(-head=>[
Link({-rel=>'next',
                                  -href=>'http://www.capricorn.com/s2.html'}),
Link({-rel=>'previous',
                                  -href=>'http://www.capricorn.com/s1.html'})
]
);
```

And here's how to create an HTTP-EQUIV <meta> tag:

```
print start_html(-head=>meta({-http_equiv => 'Content-Type',
-content => 'text/html'}))
```

JAVASCRIPTING: The **-script**, **-noScript**, **-onLoad**, **-onMouseOver**, **-onMouseOut** and **-onUnload** parameters are used to add JavaScript calls to your pages. **-script** should point to a block of text containing JavaScript function definitions. This block will be placed within a <script> block inside the HTML (not HTTP) header. The block is placed in the header in order to give your page a fighting chance of having all its JavaScript functions in place even if the user presses the stop button before the page has loaded completely. CGI.pm attempts to format the script in such a way that JavaScript-naive browsers will not choke on the code: unfortunately there are some browsers, such as Chimera for Unix, that get confused by it nevertheless.

The **-onLoad** and **-onUnload** parameters point to fragments of JavaScript code to execute when the page is respectively opened and closed by the browser. Usually these parameters are calls to functions defined in the **-script** field:

```
$query = CGI->new;
print header;
$JSCRIPT=<<END;
// Ask a silly question
function riddle_me_this() {
var r = prompt("What walks on four legs in the morning, " +
"two legs in the afternoon, " +
"and three legs in the evening?");
response(r);
}
// Get a silly answer
function response(answer) {
if (answer == "man")
alert("Right you are!");
else
alert("Wrong! Guess again.");
}
END
print start_html(-title=>'The Riddle of the Sphinx',
-script=>$JSCRIPT);
```

Use the **-noScript** parameter to pass some HTML text that will be displayed on browsers that do not have JavaScript (or browsers where JavaScript is turned off).

The <script> tag, has several attributes including ''type'', ''charset'' and ''src''. ''src'' allows you to keep JavaScript code in an external file. To use these attributes pass a HASH reference in the **-script** parameter containing one or more of -type, -src, or -code:

```
print $q->start_html(-title=>'The Riddle of the Sphinx',
-script=>{-type=>'JAVASCRIPT',
-src=>'/javascript/sphinx.js'}
);

print $q->(-title=>'The Riddle of the Sphinx',
-script=>{-type=>'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=>[
{ -type => 'text/javascript',
-src => '/javascript/utilities10.js'
},
{ -type => 'text/javascript',
-src => '/javascript/utilities11.js'
},
{ -type => 'text/jscript',
-src => '/javascript/utilities12.js'
},
{ -type => 'text/ecmascript',
-src => '/javascript/utilities219.js'
}
]
);
```

The option ''-language'' is a synonym for -type, and is supported for backwards compatibility.

The old-style positional parameters are as follows:

Parameters:

1. The title
2. The author's e-mail address (will create a <link rev=''MADE''> tag if present
3. A 'true' flag if you want to include a <base> tag in the header. This helps resolve relative addresses to absolute ones when the document is moved, but makes the document hierarchy non-portable. Use with care!

Other parameters you want to include in the <body> tag may be appended to these. This is a good place to put HTML extensions, such as colors and wallpaper patterns.

ENDING THE HTML DOCUMENT:

```
print $q->end_html;
```

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

CREATING A SELF-REFERENCING URL THAT PRESERVES STATE INFORMATION:

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

self_url() will return a URL, that, when selected, will reinvoke 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.

```
$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 the unprocessed query string with *query_string()*:

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

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

OBTAINING THE SCRIPT'S URL

```
$full_url = url();
$full_url = url(-full=>1); #alternative syntax
$relative_url = url(-relative=>1);
$absolute_url = url(-absolute=>1);
$url_with_path = url(-path_info=>1);
$url_with_path_and_query = url(-path_info=>1,-query=>1);
$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 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.

-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 mod_rewrite's rules have run.

MIXING POST AND URL PARAMETERS

```
$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.

CREATING STANDARD HTML ELEMENTS:

CGI.pm defines general HTML shortcut methods for many HTML tags. HTML shortcuts are named after a single HTML element and return a fragment of HTML text. Example:

```
print $q->blockquote(
"Many years ago on the island of",
                    },"Crete")," -P $q->a({href=>" -- http://crete.org/
"there lived a Minotaur named",
$q->strong("Fred."),
),
$q->hr;
```

This results in the following HTML code (extra newlines have been added for readability):

```
<blockquote>
Many years ago on the island of
  <a >Crete</a>" -P href=" -- http://crete.org/
there lived
a minotaur named <strong>Fred.</strong>
</blockquote>
<hr>
```

If you find the syntax for calling the HTML shortcuts awkward, you can import them into your namespace and dispense with the object syntax completely (see the next section for more details):

```
use CGI ':standard';
print blockquote(
"Many years ago on the island of",
     },"Crete")," -P a({href=>" -- http://crete.org/
"there lived a minotaur named",
strong("Fred."),
),
hr;
```

PROVIDING ARGUMENTS TO HTML SHORTCUTS

The HTML methods will accept zero, one or multiple arguments. If you provide no arguments, you get a single tag:

```
print hr; # <hr>
```

If you provide one or more string arguments, they are concatenated together with spaces and placed between opening and closing tags:

```
print h1("Chapter","1"); # <h1>Chapter 1</h1>"
```

If the first argument is a hash reference, then the keys and values of the hash become the HTML tag's attributes:

```
print a({-href=>'fred.html',-target=>'_new'},
"Open a new frame");

<a href="fred.html",target="_new">Open a new frame</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. 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 accommodate those who want to create tags of the form <img alt=>. The difference is shown in these two pieces of code:

```
CODE RESULT
img({alt=>undef}) <img alt>
img({alt=>''}) <img alt="">
```

THE DISTRIBUTIVE PROPERTY OF HTML SHORTCUTS

One of the cool features of the HTML shortcuts is that they 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>
```

This is extremely useful for creating tables. For example:

```
print table({-border=>undef},
caption('When Should You Eat Your Vegetables?'),
Tr({-align=>'CENTER',-valign=>'TOP'},
[
th(['Vegetable', 'Breakfast','Lunch','Dinner']),
td(['Tomatoes' , 'no', 'yes', 'yes']),
td(['Broccoli' , 'no', 'no', 'yes']),
td(['Onions' , 'yes','yes', 'yes'])
]
)
);
```

HTML SHORTCUTS AND LIST INTERPOLATION

Consider this bit of code:

```
print blockquote(em('Hi'),'mom!'));
```

It will ordinarily return the string that you probably expect, namely:

```
<blockquote><em>Hi</em> mom!</blockquote>
```

Note the space between the element "Hi" and the element "mom!". CGI.pm puts the extra space there using array interpolation, which is controlled by the magic $" variable. Sometimes this extra space is not what you want, for example, when you are trying to align a series of images. In this case, you can simply change the value of $" to an empty string.

```
{
local($") = '';
print blockquote(em('Hi'),'mom!'));
}
```

I suggest you put the code in a block as shown here. Otherwise the change to $ will affect all subsequent code until you explicitly reset it.

NON-STANDARD HTML SHORTCUTS

A few HTML tags don't follow the standard pattern for various reasons.

comment() generates an HTML comment (<!-- comment -->). Call it like

```
print comment('here is my comment');
```

Because of conflicts with built-in Perl functions, the following functions begin with initial caps:

```
Select
Tr
Link
Delete
Accept
Sub
```

In addition, *start_html()*, *end_html()*, *start_form()*, *end_form()*, *start_multipart_form()* and all the fill-out form tags are special. See their respective sections.

AUTOESCAPING HTML

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

`$escaped_string` = escapeHTML("unescaped string");
Escape HTML formatting characters in a 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 "<", ">" becomes ">", "&" becomes "&", and the quote character becomes """. In addition, the hexadecimal 0x8b and 0x9b characters, which some browsers incorrectly interpret as the left and right angle-bracket characters, are replaced by their numeric character entities ("&#8249" and "›"). 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.

`escapeHTML()` expects the supplied string to be a character string. This means you should Encode::decode data received from "outside" and Encode::encode your strings before sending them back outside. If your source code UTF-8 encoded and you want to upgrade string literals in your source to character strings, you can use "use utf8". See perlunitut(1), perlunifaq(1) and perlunicode(1) for more information on how Perl handles the difference between bytes and characters.

The automatic escaping does not apply to other shortcuts, such as *h1()*. You should call *escapeHTML()* yourself on untrusted data in order to protect your pages against nasty tricks that people may enter into guestbooks, etc.. To change the character set, use *charset()*. To turn autoescaping off completely, use *autoEscape(0)*:

`$charset` = charset([$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.

CREATING FILL-OUT FORMS:

*General note*The v arious form-creating methods all return strings to the caller, containing the tag or tags 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.

*Another note*The default v alues that you specify for the forms are only used the **first** time the script is invoked (when there is no query string). On subsequent invocations of the script (when there is a query string), the former values are used even if they are 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 (a new feature in version 2.15). This forces the default value to be used, regardless of the previous value:

```
print textfield(-name=>'field_name',
-default=>'starting value',
-override=>1,
-size=>50,
-maxlength=>80);
```

*Yet another note*By default, the text and labels of form elemen ts are escaped according to HTML rules. This means that you can safely use "<CLICK ME>" as the label for a button. However, it also interferes with your ability to incorporate special HTML character sequences, such as 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 = CGI->new;
$query->autoEscape(0);
```

Note that *autoEscape()* is exclusively used to effect the behavior of how some CGI.pm HTML generation functions handle escaping. Calling *escapeHTML()* explicitly will always escape the HTML.

A Lurking Trap! Some of the form-element generating methods return multiple tags. In a scalar context, the tags will be concatenated together with spaces, or whatever is the current value of the $ global. In a list context, the methods will return a list of elements, allowing you to modify them if you wish. Usually you will not notice this behavior, but beware of this:

```
printf("%s\n",end_form())
```

end_form() produces several tags, and only the first of them will be printed because the format only expects one value.

<p>

CREATING AN ISINDEX TAG

```
print isindex(-action=>$action);

-or-

print isindex($action);
```

Prints out an <isindex> tag. Not very exciting. The parameter -action specifies the URL of the script to process the query. The default is to process the query with the current script.

STARTING AND ENDING A FORM

```
print start_form(-method=>$method,
-action=>$action,
-enctype=>$encoding);
<... various form stuff ...>
print end_form;

-or-

print start_form($method,$action,$encoding);
<... various form stuff ...>
print end_form;
```

start_form() will return a <form> tag with the optional method, action and form encoding that you specify. The defaults are:

```
method: POST
action: this script
enctype: application/x-www-form-urlencoded for non-XHTML
multipart/form-data for XHTML, see multipart/form-data below.
```

end_form() returns the closing </form> tag.

Start_form()'s enctype argument tells the browser how to package the various fields of the form before sending the form to the server. Two values are possible:

Note: These methods were previously named *startform()* and *endform()*. These methods are now DEPRECATED. Please use*start_form()* and *end_form()* instead.

application/x-www-form-urlencoded
: This is the older type of encoding. 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. 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. 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 designed to handle them.

  If XHTML is activated (the default), then forms will be automatically created using this type of encoding.

The *start_form()* method uses the older form of encoding by default unless XHTML is requested. If you want to use the newer form of encoding by default, you can call ***start_multipart_form()*** instead of ***start_form()***. The method ***end_multipart_form()*** is an alias to ***end_form()***.

JAVASCRIPTING: The **-name** and **-onSubmit** parameters are provided for use with JavaScript. The -name parameter gives the form a name so that it can be identified and manipulated by JavaScript functions. -onSubmit should point to a JavaScript function that will be executed just before the form is submitted to your server. You can use this opportunity to check the contents of the form for consistency and completeness. If you find something wrong, you can put up an alert box or maybe fix things up yourself. You can abort the submission by returning false from this function.

Usually the bulk of JavaScript functions are defined in a <script> block in the HTML header and -onSubmit points to one of these function call. See *start_html()* for details.

FORM ELEMENTS

After starting a form, you will typically create one or more textfields, popup menus, radio groups and other form elements. Each of these elements takes a standard set of named arguments. Some elements also have optional arguments. The standard arguments are as follows:

-name
: The name of the field. After submission this name can be used to retrieve the field's value using the *param()* method.

-value, **-values**
: The initial value of the field which will be returned to the script after form submission. Some form elements, such as text fields, take a single scalar -value argument. Others, such as popup menus, take a reference to an array of values. The two arguments are synonyms.

-tabindex
: A numeric value that sets the order in which the form element receives focus when the user presses the tab key. Elements with lower values receive focus first.

-id A string identifier that can be used to identify this element to JavaScript and DHTML.

-override
A boolean, which, if true, forces the element to take on the value specified by **-value**, overriding the sticky behavior described earlier for the **-nosticky** pragma.

-onChange, **-onFocus**, **-onBlur**, **-onMouseOver**, **-onMouseOut**, **-onSelect**
These are used to assign JavaScript event handlers. See the JavaScripting section for more details.

Other common arguments are described in the next section. In addition to these, all attributes described in the HTML specifications are supported.

CREATING A TEXT FIELD

```
print textfield(-name=>'field_name',
-value=>'starting value',
-size=>50,
-maxlength=>80);
-or-

print textfield('field_name','starting value',50,80);
```

textfield() will return a text input field.

Parameters

1. The first parameter is the required name for the field (-name).
2. The optional second parameter is the default starting value for the field contents (-value, formerly known as -default).
3. The optional third parameter is the size of the field in characters (-size).
4. The optional fourth parameter is the maximum number of characters the field will accept (-maxlength).

As with all these methods, the field will be initialized with its previous contents from earlier invocations of the script. When the form is processed, the value of the text field can be retrieved with:

```
$value = param('foo');
```

If you want to reset it from its initial value after the script has been called once, you can do so like this:

```
param('foo',"I'm taking over this value!");
```

CREATING A BIG TEXT FIELD

```
print textarea(-name=>'foo',
-default=>'starting value',
-rows=>10,
-columns=>50);

-or

print 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.

CREATING A PASSWORD FIELD

```
print password_field(-name=>'secret',
-value=>'starting value',
-size=>50,
-maxlength=>80);
-or-

print 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

```
print filefield(-name=>'uploaded_file',
-default=>'starting value',
-size=>50,
-maxlength=>80);
-or-

print filefield('uploaded_file','starting value',50,80);
```

filefield() will return a file upload field. In order to take full advantage of this *you must use the new multipart encoding scheme*for the form. Y ou can do this either by calling ***start_form()*** with an encoding type of **&CGI::MULTIPART**, or by calling the new method ***start_multipart_form()*** instead of vanilla ***start_form()***.

Parameters

1. The first parameter is the required name for the field (-name).
2. The optional second parameter is the starting value for the field contents to be used as the default file name (-default).

   For security reasons, browsers don't pay any attention to this field, and so the starting value will always be blank. Worse, the field loses its ''sticky'' behavior and forgets its previous contents. The starting value field is called for in the HTML specification, however, and possibly some browser will eventually provide support for it.
3. The optional third parameter is the size of the field in characters (-size).
4. The optional fourth parameter is the maximum number of characters the field will accept (-maxlength).

JAVASCRIPTING: The **-onChange**, **-onFocus**, **-onBlur**, **-onMouseOver**, **-onMouseOut** and **-onSelect** parameters are recognized. See *textfield()* for details.

PROCESSING A FILE UPLOAD FIELD

Basics

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

```
$lightweight_fh = $q->upload('field_name');

# undef may be returned if it's not a valid file handle
if (defined $lightweight_fh) {
# Upgrade the handle to one compatible with IO::Handle:
my $io_handle = $lightweight_fh->handle;

open (OUTFILE,'>>','/usr/local/web/users/feedback');
while ($bytesread = $io_handle->read($buffer,1024)) {
print OUTFILE $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()*:

```
$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.

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

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:

```
$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.

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:

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

You are free to create a custom HTML page to complain about the error, if you wish.

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.

```
$q = CGI->new(\&hook [,$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::Handle. 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::Handle. It's a big win for compatibility for a small penalty of loading IO::Handle the first time you call it.

CREATING A POPUP MENU

```
print popup_menu('menu_name',
['eenie','meenie','minie'],
'meenie');

-or-

%labels = ('eenie'=>'your first choice',
'meenie'=>'your second choice',
```

```
'minie'=>'your third choice');
%attributes = ('eenie'=>{'class'=>'class of first choice'});
print popup_menu('menu_name',
['eenie','meenie','minie'],
'meenie',\%labels,\%attributes);

-or (named parameter style)-

print popup_menu(-name=>'menu_name',
-values=>['eenie','meenie','minie'],
-default=>['meenie','minie'],
-labels=>\%labels,
-attributes=>\%attributes);
```

popup_menu() creates a menu.

1. The required first argument is the menu's name (-name).
2. 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''.
3. The optional third parameter (-default) is the name of the default menu choice. If not specified, the first item will be the default. The values of the previous choice will be maintained across queries. Pass an array reference to select multiple defaults.
4. The optional fourth parameter (-labels) is provided for people who want to use different values for the user-visible label inside the popup menu and the value returned to your script. It's a pointer to an hash relating menu values to user-visible labels. If you leave this parameter blank, the menu values will be displayed by default. (You can also leave a label undefined if you want to).
5. The optional fifth parameter (-attributes) is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

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

```
$popup_menu_value = param('menu_name');
```

CREATING AN OPTION GROUP

Named parameter style

```
print popup_menu(-name=>'menu_name',
-values=>[qw/eenie meenie minie/,
optgroup(-name=>'optgroup_name',
-values => ['moe','catch'],
-attributes=>{'catch'=>{'class'=>'red'}})],
-labels=>{'eenie'=>'one',
'meenie'=>'two',
'minie'=>'three'},
-default=>'meenie');

Old style
print popup_menu('menu_name',
['eenie','meenie','minie',
optgroup('optgroup_name', ['moe', 'catch'],
{'catch'=>{'class'=>'red'}})],'meenie',
{'eenie'=>'one','meenie'=>'two','minie'=>'three'});
```

optgroup() creates an option group within a popup menu.

1. The required first argument (**-name**) is the label attribute of the optgroup and is **not** inserted in the parameter list of the query.
2. 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).
3. The optional third parameter (**-labels**) allows you to pass a reference to a hash 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.
4. An optional fourth parameter (**-labeled**) can be set to a true value and indicates that the values should be used as the label attribute for each option element within the optgroup.
5. An optional fifth parameter (-novals) can be set to a true value and indicates to suppress the val attribute in each option element within the optgroup.

   See the discussion on optgroup at W3C (http://www.w3.org/TR/REC-html40/interact/forms.html#edef-OPTGROUP) for details.
6. An optional sixth parameter (-attributes) is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

CREATING A SCROLLING LIST

```
print scrolling_list('list_name',
['eenie','meenie','minie','moe'],
['eenie','moe'],5,'true',{'moe'=>{'class'=>'red'}});
-or-

print scrolling_list('list_name',
['eenie','meenie','minie','moe'],
['eenie','moe'],5,'true',
\%labels,%attributes);

-or-

print scrolling_list(-name=>'list_name',
-values=>['eenie','meenie','minie','moe'],
-default=>['eenie','moe'],
-size=>5,
-multiple=>'true',
-labels=>\%labels,
-attributes=>\%attributes);
```

scrolling_list() creates a scrolling list.

Parameters:

1. The first and second arguments are the list name (-name) and values (-values). As in the popup menu, the second argument should be an array reference.
2. 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. In the named parameter version, you can use the synonym ''-defaults'' for this parameter.
3. The optional fourth argument is the size of the list (-size).

4. The optional fifth argument can be set to true to allow multiple simultaneous selections (-multiple). Otherwise only one selection will be allowed at a time.
5. The optional sixth argument is a pointer to a hash containing long user-visible labels for the list items (-labels). If not provided, the values will be displayed.
6. The optional sixth parameter (-attributes) is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

   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 = param('list_name');
```

CREATING A GROUP OF RELATED CHECKBOXES

```
print checkbox_group(-name=>'group_name',
-values=>['eenie','meenie','minie','moe'],
-default=>['eenie','moe'],
-linebreak=>'true',
-disabled => ['moe'],
-labels=>\%labels,
-attributes=>\%attributes);

print checkbox_group('group_name',
['eenie','meenie','minie','moe'],
['eenie','moe'],'true',\%labels,
{'moe'=>{'class'=>'red'}});

HTML3-COMPATIBLE BROWSERS ONLY:

print 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.

Parameters:

1. The first and second arguments are the checkbox name and values, respectively (-name and -values). As in the popup menu, the second argument should be an array 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.
2. 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.
3. 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.

The optional **-labels** argument is a pointer to a hash relating the checkbox values to the user-visible labels that will be printed next to them. If not provided, the values will be used as the default.

The optional parameters **-rows**, and **-columns** 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.

The option **-disabled** takes an array of checkbox values and disables them by greying them out

(this may not be supported by all browsers).

The optional **-attributes** argument is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

The optional **-tabindex** argument can be used to control the order in which radio buttons receive focus when the user presses the tab button. If passed a scalar numeric value, the first element in the group will receive this tab index and subsequent elements will be incremented by one. If given a reference to an array of radio button values, then the indexes will be jiggered so that the order specified in the array will correspond to the tab order. You can also pass a reference to a hash in which the hash keys are the radio button values and the values are the tab indexes of each button. Examples:

```
-tabindex => 100 # this group starts at index 100 and counts up
-tabindex => ['moe','minie','eenie','meenie'] # tab in this order
-tabindex => {meenie=>100,moe=>101,minie=>102,eenie=>200} # tab in this order
```

The optional **-labelattributes** argument will contain attributes attached to the <label> element that surrounds each button.

When the form is processed, 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:

```
@turned_on = param('group_name');
```

The value returned by *checkbox_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

```
@h = checkbox_group(-name=>'group_name',-values=>\@values);
&use_in_creative_way(@h);
```

CREATING A STANDALONE CHECKBOX

```
print checkbox(-name=>'checkbox_name',
-checked=>1,
-value=>'ON',
-label=>'CLICK ME');

-or-

print checkbox('checkbox_name','checked','ON','CLICK ME');
```

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

Parameters:

1. The first parameter is the required name for the checkbox (-name). It will also be used for the user-readable label printed next to the checkbox.
2. The optional second parameter (-checked) specifies that the checkbox is turned on by default. Synonyms are -selected and -on.
3. The optional third parameter (-value) specifies the value of the checkbox when it is checked. If not provided, the word ''on'' is assumed.
4. The optional fourth parameter (-label) is the user-readable label to be attached to the checkbox. If not provided, the checkbox name is used.

The value of the checkbox can be retrieved using:

```
$turned_on = param('checkbox_name');
```

CREATING A RADIO BUTTON GROUP

```
print radio_group(-name=>'group_name',
-values=>['eenie','meenie','minie'],
-default=>'meenie',
-linebreak=>'true',
-labels=>\%labels,
-attributes=>\%attributes);

-or-

print radio_group('group_name',['eenie','meenie','minie'],
'meenie','true',\%labels,\%attributes);


HTML3-COMPATIBLE BROWSERS ONLY:

print 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)

Parameters:

1. The first argument is the name of the group and is required (-name).
2. 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’’.
3. The optional third parameter (-default) is the name of the default button to turn on. If not specified, the first item will be the default. You can provide a nonexistent button name, such as ‘‘-’’ to start up with no buttons selected.
4. The optional fourth parameter (-linebreak) can be set to ’true’ to put line breaks between the buttons, creating a vertical list.
5. The optional fifth parameter (-labels) is a pointer to an associative array relating the radio button values to user-visible labels to be used in the display. If not provided, the values themselves are displayed.

All modern browsers can take advantage of the optional parameters **-rows**, and **-columns**. These parameters cause *radio_group()* to return an HTML3 compatible table containing the radio group 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 **-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 interpretation of the radio buttons — they’re still a single named unit.

The optional **-tabindex** argument can be used to control the order in which radio buttons receive focus when the user presses the tab button. If passed a scalar numeric value, the first element in the group will receive this tab index and subsequent elements will be incremented by one. If given a reference to an array of radio button values, then the indexes will be jiggered so that the order specified in the array will correspond to the tab order. You can also pass a reference to a hash in which the hash keys are the radio button values and the values are the tab indexes of each button. Examples:

```
-tabindex => 100 # this group starts at index 100 and counts up
-tabindex => ['moe','minie','eenie','meenie'] # tab in this order
-tabindex => {meenie=>100,moe=>101,minie=>102,eenie=>200} # tab in this order
```

The optional **-attributes** argument is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

The optional **-labelattributes** argument will contain attributes attached to the <label> element that surrounds each button.

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

```
$which_radio_button = param('group_name');
```

The value returned by *radio_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

```
@h = radio_group(-name=>'group_name',-values=>\@values);
&use_in_creative_way(@h);
```

CREATING A SUBMIT BUTTON

```
print submit(-name=>'button_name',
-value=>'value');

-or-

print submit('button_name','value');
```

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

Parameters:

1. 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.
2. The second argument (-value) is also optional. This gives the button a value that will be passed to your script in the query string. The name will also be used as the user-visible label.
3. 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.

You can figure out which button was pressed by using different values for each one:

```
$which_one = param('button_name');
```

CREATING A RESET BUTTON

```
print reset
```

reset() creates the "reset" button. Note that it restores the form to its value from the last time the script was called, NOT necessarily to the defaults.

Note that this conflicts with the Perl *reset()* built-in. Use *CORE::reset()* to get the original reset function.

CREATING A DEFAULT BUTTON

```
print defaults('button_label')
```

defaults() creates a button that, when invoked, will cause the form to be completely reset to its defaults, wiping out all the changes the user ever made.

CREATING A HIDDEN FIELD

```
print hidden(-name=>'hidden_name',
-default=>['value1','value2'...]);

-or-
```

```
print 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.

Parameters:

1. The first argument is required and specifies the name of this field (-name).
2. The second argument is also required and specifies its value (-default). In the named parameter style of calling, you can provide a single value here or a reference to a whole list

Fetch the value of a hidden field this way:

```
$hidden_value = param('hidden_name');
```

Note, that 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:

```
param('hidden_name','new','values','here');
```

CREATING A CLICKABLE IMAGE BUTTON

```
print image_button(-name=>'button_name',
-src=>'/source/URL',
-align=>'MIDDLE');

-or-

print image_button('button_name','/source/URL','MIDDLE');
```

image_button() produces a clickable image. When it's clicked on the position of the click is returned to your script as ''button_name.x'' and ''button_name.y'', where ''button_name'' is the name you've assigned to it.

Parameters:

1. The first argument (-name) is required and specifies the name of this field.
2. The second argument (-src) is also required and specifies the URL
3. The third option (-align, optional) is an alignment type, and may be TOP, BOTTOM or MIDDLE

Fetch the value of the button this way: `$x` = param('button_name.x'); `$y` = param('button_name.y');

CREATING A JAVASCRIPT ACTION BUTTON

```
print button(-name=>'button_name',
-value=>'user visible label',
-onClick=>"do_something()");

-or-

print button('button_name',"user visible value","do_something()");
```

button() produces an `<input>` tag with `type="button"`. When it's pressed the fragment of JavaScript code pointed to by the **-onClick** parameter will be executed.

HTTP COOKIES

Browsers 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:

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:

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

```
$cookie=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 CGI ':standard';
print header(-cookie=>$my_cookie);
```

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

```
$cookie1 = cookie(-name=>'riddle_name',
-value=>"The Sphynx's Question");
$cookie2 = cookie(-name=>'answers',
-value=>\%answers);
print header(-cookie=>[$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:

```
use CGI;
$query = CGI->new;
$riddle = $query->cookie('riddle_name');
%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
$c=cookie(-name=>'answers',-value=>[param('answers')]);
# vice-versa
param(-name=>'answers',-value=>[cookie('answers')]);
```

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

```
@cookies = cookie();
```

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

WORKING WITH FRAMES

It's possible for CGI.pm scripts to write into several browser panels and windows using the HTML 4 frame mechanism. There are three techniques for defining new frames programmatically:

1. Create a <Frameset> document
   After writing out the HTTP header, instead of creating a standard HTML document using the *start_html()* call, create a <frameset> document that defines the frames on the page. Specify your script(s) (with appropriate parameters) as the SRC for each of the frames.

   There is no specific support for creating <frameset> sections in CGI.pm, but the HTML is very simple to write.

2. Specify the destination for the document in the HTTP header
   You may provide a **-target** parameter to the *header()* method:

```
print header(-target=>'ResultsWindow');
```

This will tell the browser to load the output of your script into the frame named "ResultsWindow". If a frame of that name doesn't already exist, the browser will pop up a new window and load your script's document into that. There are a number of magic names that you can use for targets. See the HTML `<frame>` documentation for details.

3. Specify the destination for the document in the <form> tag
   You can specify the frame to load in the FORM tag itself. With CGI.pm it looks like this:

```
print start_form(-target=>'ResultsWindow');
```

When your script is reinvoked by the form, its output will be loaded into the frame named "ResultsWindow". If one doesn't already exist a new window will be created.

The script "frameset.cgi" in the examples directory shows one way to create pages in which the fill-out form and the response live in side-by-side frames.

SUPPORT FOR JAVASCRIPT

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 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()* 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 start_form(-onSubmit=>‘‘validateMe(this)’’);

See the javascript.cgi script for a demonstration of how this all works.

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 treated as the source URL for the stylesheet, 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 type of the stylesheet by adding the optional **-type** parameter to the hash pointed to by **-style**. If not specified, the style 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 have the ***span()*** method available. Here's a quick and dirty example of using CSS's. See the CSS specification at http://www.w3.org/Style/CSS/ for more information.

```
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 **-code** or **-src** 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 start_html (-style => {-verbatim => '@import url("/server-common/css/'.$cssFile.'");', -src => '/server-common/css/core.css'});

This will generate an HTML header that contains 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"/>
```

<p>

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})
```

To create primary and "alternate" stylesheet, use the **-alternate** option:

```
start_html(-style=>{-src=>[
{-src=>'/styles/print.css'},
{-src=>'/styles/alt.css',-alternate=>1}
]
});
```

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
```

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 Dump
```

Produces something that looks like:

```
<ul>
<li>name1
<ul>
<li>value1
<li>value2
</ul>
<li>name2
<ul>
<li>value1
</ul>
</ul>
```

As a shortcut, you can interpolate the entire CGI object into a string and it will be replaced with the a nice HTML dump shown above:

```
$query=CGI->new;
print "<h2>Current Values</h2> $query\n";
```

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.

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*p ath_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.

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

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. Not available for all browsers.

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.

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. This only works with older browsers such as Mosaic. Newer browsers do not report the user name for privacy reasons!

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:

```
$requested_language = http('Accept-language');
$requested_language = http('Accept_language');
$requested_language = 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. However, note that if you have applied Service Pack 6, much of the functionality of NPH scripts, including the ability to redirect while setting a cookie, **do not work at all**on IIS without a special patc h from Microsoft. See http://web.archive.org/web/20010812012030/http://support.microsoft.com/support/kb/articles/Q280/3/41.AS Non-Parsed Headers Stripped From CGI Applications That Have nph- Prefix in Name.

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/local/bin/perl
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);
```

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.

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:

```
use CGI qw/:standard/;
use CGI::Carp 'fatalsToBrowser';
$CGI::POST_MAX=1024 * 100; # max 100K posts
$CGI::DISABLE_UPLOADS = 1; # no uploads
```

2. 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()*.

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;
}
```

However it isn't clear that any browser currently knows what to do with this status code. It might be better just to create an HTML 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:

```
$q = $in{CGI};
print $q->textfield(-name=>'wow',
-value=>'does this really work?');
```

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()
HtmlTop()
HtmlBot()
SplitParam()
MethGet()
MethPost()
```

Cgi-lib functions that are not available in CGI.pm

```
* Extended form of ReadParse()
The extended form of ReadParse() that provides for file upload
spooling, is not available.

* MyBaseURL()
This function is not available. Use CGI.pm's url() method instead.

* MyFullURL()
This function is not available. Use CGI.pm's self_url() method
instead.

* CgiError(), CgiDie()
These functions are not supported. Look at CGI::Carp for the way I
prefer to handle error messages.

* PrintVariables()
This function is not available. To achieve the same effect,
just print out the CGI object:
```

```
use CGI qw(:standard);
$q = CGI->new;
print h1("The Variables Are"),$q;

* PrintEnv()
This function is not available. You'll have to roll your own if you really need it.
```

AUTHOR INFORMATION

Address bug reports and comments to: 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.

CREDITS

Thanks very much to:

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.

A COMPLETE EXAMPLE OF A SIMPLE FORM-BASED SCRIPT

```
#!/usr/local/bin/perl

use CGI ':standard';

print header;
print start_html("Example CGI.pm Form");
print "<h1> Example CGI.pm Form</h1>\n";
print_prompt();
do_work();
print_tail();
print end_html;
```

```
sub print_prompt {
print start_form;
print "<em>What's your name?</em><br>";
print textfield('name');
print checkbox('Not my real name');

print "<p><em>Where can you find English Sparrows?</em><br>";
print checkbox_group(
-name=>'Sparrow locations',
-values=>[England,France,Spain,Asia,Hoboken],
-linebreak=>'yes',
-defaults=>[England,Asia]);

print "<p><em>How far can they fly?</em><br>",
radio_group(
-name=>'how far',
-values=>['10 ft','1 mile','10 miles','real far'],
-default=>'1 mile');

print "<p><em>What's your favorite color?</em> ";
print popup_menu(-name=>'Color',
-values=>['black','brown','red','yellow'],
-default=>'red');

print hidden('Reference','Monty Python and the Holy Grail');

print "<p><em>What have you got there?</em><br>";
print scrolling_list(
-name=>'possessions',
-values=>['A Coconut','A Grail','An Icon',
'A Sword','A Ticket'],
-size=>5,
-multiple=>'true');

print "<p><em>Any parting comments?</em><br>";
print textarea(-name=>'Comments',
-rows=>10,
-columns=>50);

print "<p>",reset;
print submit('Action','Shout');
print submit('Action','Scream');
print end_form;
print "<hr>\n";
}

sub do_work {

print "<h2>Here are the current settings in this form</h2>";

for my $key (param) {
print "<strong>$key</strong> -> ";
my @values = param($key);
```

```
print join(", ",@values),"<br>\n";
}
}

sub print_tail {
print <<END;
<hr>
<address>Lincoln D. Stein</address><br>
<a href="/">Home Page</a>
END
}
```

BUGS

Please report them.

SEE ALSO

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

CGI::Fast - supports running CGI applications under FastCGI

CGI::Pretty - pretty prints HTML generated by CGI.pm (with a performance penalty)