COMP 2021

Unix and Script Programming

CGI Programming in Perl (2)

CGI Apps vs. Java Applets

- CGI is a specification that can be used by any programming language
- CGI application
  - run on a Web server
  - Great for searching databases, processing HTML form data, and other applications that require limited interaction with a user.
  - Can be as large as needed because they reside and executed on the Web server
- Java application are good when you need a high degree of interaction with users, e.g. games or animation
  - Relatively small because they're transmitted through the Internet to the client
- A design of with both Java and CGI application
  - Client side: Java applet validate the field input of a form
  - Server side: CGI run query to search database

Passing Information using GET Method

- GET and/or POST method help to pass some information from your browser to web server and ultimately to your CGI program
- The GET method sends the encoded user information
  - `http://ihome.ust.hk/~lixin/cgi-bin/get.cgi?key1=value1&key2=value2`
  - Default method to pass information
  - Produce a long string that appears in browser's Location:box
  - At most 1024 character in the request string
  - Information accessible in your CGI Program through QUERY_STRING environment variable
  - Never use if you have password or other sensitive information

Query String Value

Firefox | Environment

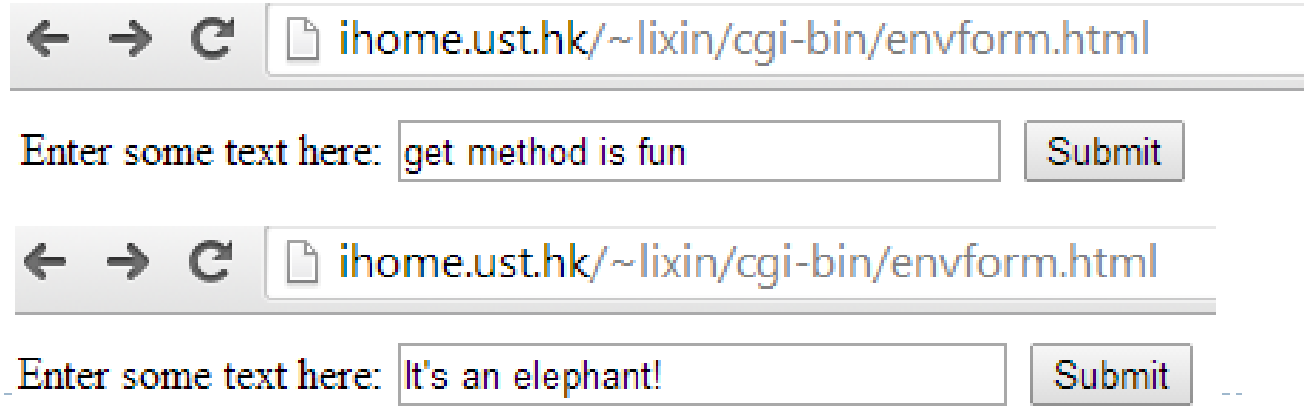
ihome.ust.hk/~lixin/cgi-bin/env.cgi?key1=value1&key2=value2&key3=value3

DOCUMENT_ROOT = /usr/local/apache/htdocs
GATEWAY_INTERFACE = CGI/1.1
HTTP_ACCEPT = text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
HTTP_ACCEPT_ENCODING = gzip, deflate
HTTP_ACCEPT_LANGUAGE = en-US,en;q=0.5
HTTP_CACHE_CONTROL = max-age=259200
HTTP_CONNECTION = keep-alive
HTTP_COOKIE = pgv_pvi=4995362816; __utma=104496004.1798391110.1384150517.1:
(not%20provided); _ga=GA1.2.1798391110.1384150517
HTTP_HOST = ihome.ust.hk
HTTP_USER_AGENT = Mozilla/5.0 (Windows NT 6.1; WOW64; rv:28.0) Gecko/2010010
HTTP_VIA = 1.1 proxy.cse.ust.hk:8080 (squid/2.6.STABLE21)
HTTP_X_FORWARDED_FOR = 143.89.128.81
PATH = /usr/local/bin:/usr/bin:/bin
QUERY_STRING = key1=value1&key2=value2&key3=value3
REMOTE_ADDR = 143.89.41.238
REMOTE_HOST = csfw3.cse.ust.hk
REMOTE_PORT = 57674
REQUEST_METHOD = GET
REQUEST_URI = /~lixin/cgi-bin/env.cgi?key1=value1&key2=value2&key3=value3
SCRIPT_FILENAME = /home/lixin/public_html/cgi-bin/env.cgi
SCRIPT_NAME = /~lixin/cgi-bin/env.cgi

Simple HTML Form using GET

- envform.html

```
<html><head><title>Test Form</title></head>
<body>

<form action="http://ihome.ust.hk/~lixin/cgi-bin/env.cgi"
method="GET">
Enter some text here:
<input type="text" name="sample_text" size=30>
<input type="submit"><p>
</form>

</body></html>
```



CGI to Process User's Input

- **env.cgi**

```
#!/usr/local/bin/perl5 -w
use strict;
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

print header;
print start_html("Environment");

foreach my $key (sort(keys(%ENV))) {
    print "$key = $ENV{$key}<br>\n";
}

print end_html;
```

User's Input with GET Method

ihome.ust.hk/~lixin/cgi-bin/env.cgi?sample_text=get+method+is+fun

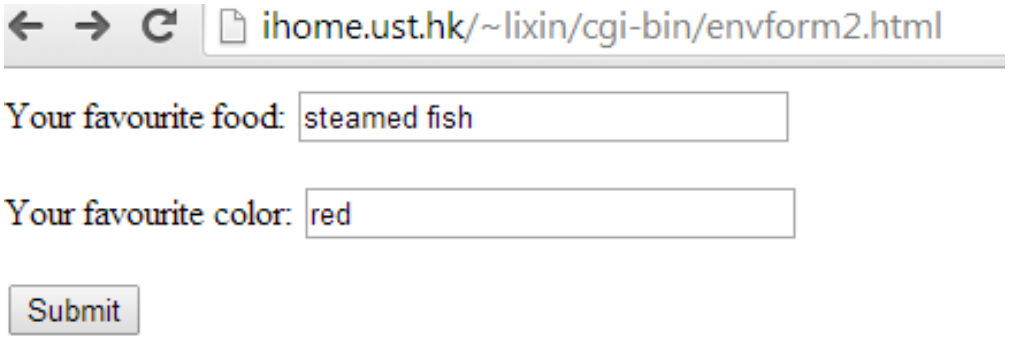
ihome.ust.hk/~lixin/cgi-bin/env.cgi?sample_text=It%27s+an+elephant%21

- Value for QUERY_STRING looks like
  - Name_of_field=whatever+you+typed
  - Any spaces in the string you typed have been replaced with a +
  - URL-encoding: various punctuation and other special non-alphanumeric characters have been replaced with a %-code.

Multiple Fields to Send

- envform2.html

```
<html><head><title>Test Form 2</title></head>
<body>

<form action="http://ihome.ust.hk/~lixin/cgi-bin/env.cgi"
method="GET">
Your favourite food: <input type="text" name="food"
size=30><p>
Your favourite color:  <input type="text" name="color"
size=30><p>
<input type="submit">
</form>

</body></html>
```



Parse User's Input

ihome.ust.hk/~lixin/cgi-bin/env.cgi?food=steamed+fish&color=red

- `env.cgi` program receives

```
$ENV{QUERY_STRING} = "food=steamed+fish&color=red"
```

- The two form values are separated by an ampersand (&). You can divide the query string with Perl's split function

```
my @values = split(/&/,$ENV{QUERY_STRING});
```

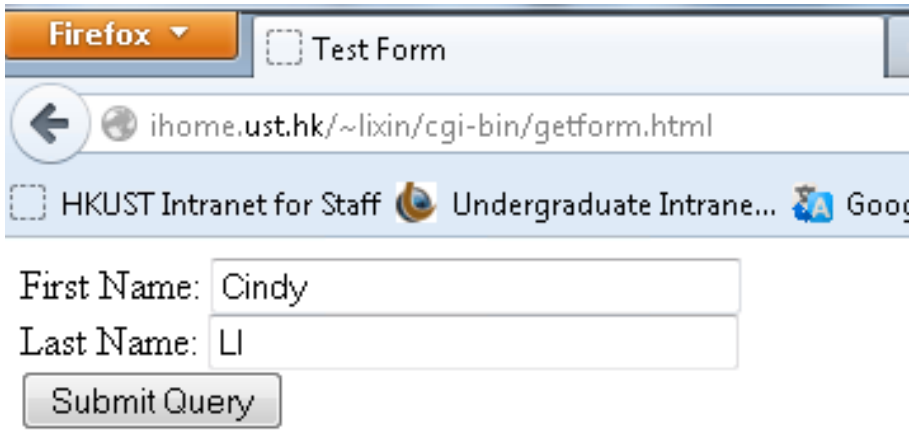
- We can further split each string on the "=" character using a foreach loop:

```
foreach my $i (@values) {
    my($fieldname, $data) = split(/=/, $i);
    print "$fieldname = $data<br>\n";
}
```

棘手

Troublesome

Using CGI.pm to Parse the Query String



firstname = Cindy

lastname = LI

Collect User's input via a Form

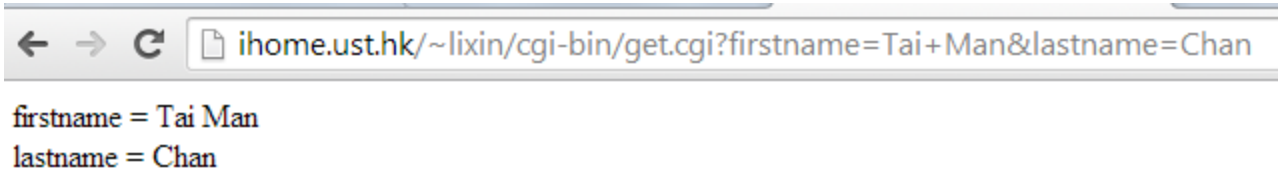
- **getform.html**

```
<html><head><title>Test Form</title></head>
<body>

<form action="http://ihome.ust.hk/~lixin/cgi-bin/get.cgi"
method="GET">
First Name: <input type="text" name="firstname"
size=30><br>
Last Name: <input type="text" name="lastname" size=30><br>
<input type="submit"><p>
</form>

</body></html>
```

Process User Inputs

- **get.cgi**

```
#!/usr/local/bin/perl5 -w
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Get Form");

my %form;
foreach my $p (param()) {
    $form{$p} = param($p);
    print "$p = $form{$p}<br>\n";
}
print end_html;
```



More on param

- `param` is NOT a Variable, it's a function call

- You can't do this: `print "$p = param($p)<br>\n";`
- `print param($p);` prints the value of `param($p)`
- Or call `param` outside of the double-quoted strings:

```
print "$p = ", param($p), "<br>\n";
```

- Can't use `param('fieldname')` inside a here-document.

- It's easier to assign the form values to individual variables:

```
my $firstname = param('firstname');
my $lastname = param('lastname');
```

- Another way would be to assign every form value to a hash

```
my(%form);
foreach my $p (param()) {
    $form{$p} = param($p);
}
```

POST Method

- Most forms send their data using the POST method.
  - More secure than GET
  - Data isn't sent as part of the URL
  - Can send more data with POST.
- Web browser encodes form data when sending out
  - URL encoding: alphanumeric characters are sent as themselves; spaces are converted to plus signs (+); other characters — like tabs, quotes, etc. — are converted to "%HH" — a percent sign and two hexadecimal digits representing the ASCII code of the character.
- Your program must decode these.
  - CGI.pm module does this work for you.
  - You access the decoded form values the same way you did with GET `$value = param('fieldname');`

Old Ways of Decoding Form Data

- Before CGI.pm was bundled with Perl, CGI programmers had to write their own form-parsing code.
- From older CGI books or old code

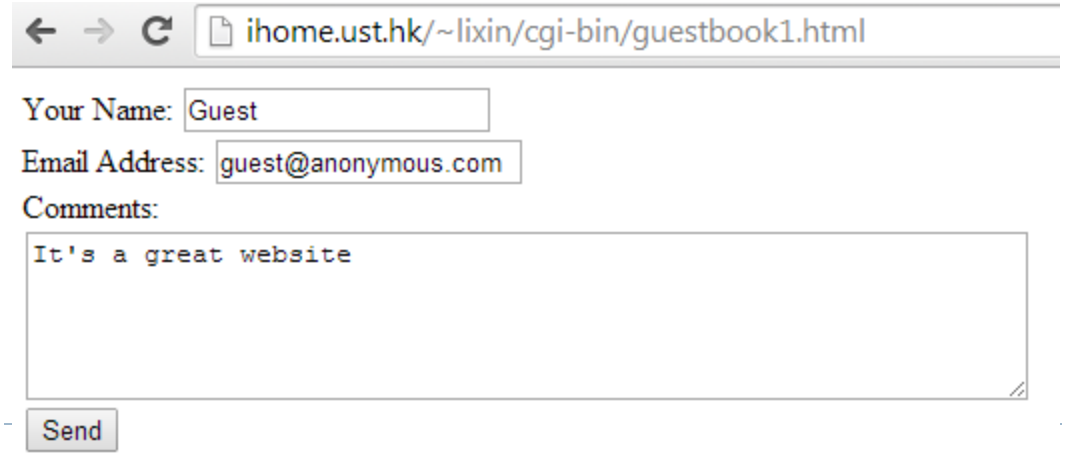
```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
```

- reads the posted form data from standard input, loops through the fieldname=value fields in the form, and uses the pack function to do URL-decoding. Then it stores each fieldname/value pair in a hash called %FORM.

User Input with Post Method: Guestbook Form

- Guestbook1.html

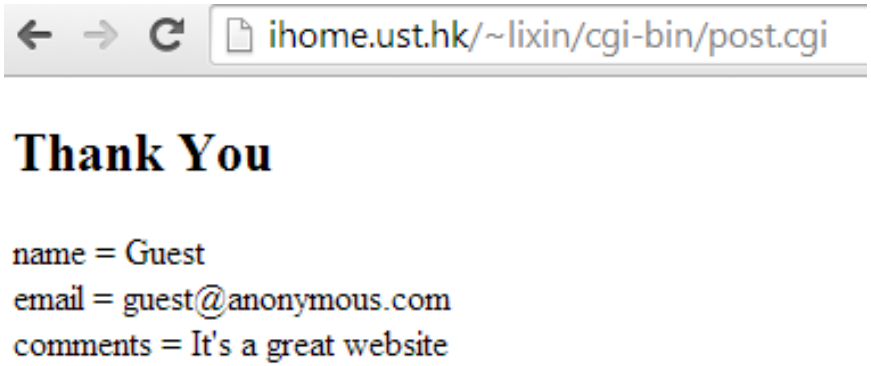
```
<form action="http://ihome.ust.hk/~lixin/cgi-bin/post.cgi"
method="POST">
Your Name: <input type="text" name="name"><br>
Email Address: <input type="text" name="email"><br>
Comments:<br>
<textarea name="comments" rows="5"
   cols="60"></textarea><br>
<input type="submit" value="Send">
</form>
```



Process the Form Data from POST Method

- **post.cgi**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;

print header;
print start_html("Thank You");
print h2("Thank You");

my %form;
foreach my $p (param()) {
    $form{$p} = param($p);
    print "$p = $form{$p}<br>\n";
}
print end_html;
```



Send Email Notification

- guestbook.cgi (part)

```
# Set the PATH environment variable to the same path
# where sendmail is located:
$ENV{PATH} = "/usr/sbin";
# open the pipe to sendmail
open (MAIL, "|/usr/sbin/sendmail -oi -t") or
    &die("Can't fork for sendmail: $!\n");
# change this to your own e-mail address
my $recipient = 'lixin@ust.hk';
# Start printing the mail headers
# You must specify who it's to, or it won't be delivered:
print MAIL "To: $recipient\n";
```

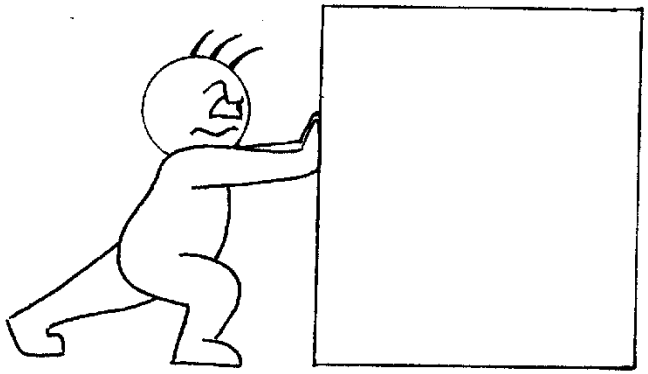
```
# From should probably be the webserver.
print MAIL "From: autosender\@COMP2021CGI\n";
# print a subject line so you know it's from your form
cgi.
print MAIL "Subject: Form Data\n\n";
# Now print the body of your mail message.
foreach my $p (param()) {
    print MAIL "$p = ", param($p), "\n";
}
# Be sure to close the MAIL input stream so that the
# message actually gets mailed.

close(MAIL);
```

Dynamic Documents

- Browsers have always been driven by user input.
  - You click on a link or an icon or an image and some data comes to you.
- Need servers to be able to push new data to the browser.
  - a businessman who wants to see new stock quotes every 5 minutes.
  - Until recently, that has not been possible.
- Two complementary methods for generating dynamic documents: *server push* and *client pull*.

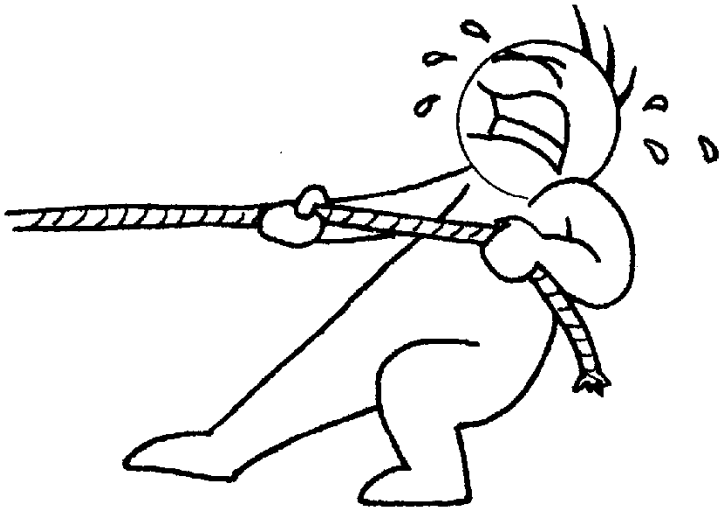
Server Push

- The server sends a chunk of data.
- The browser displays the data, but leaves the connection open.
- The server sends more data whenever it wants and the browser displays it, always leaving the connection open.



Client Pull

- The server sends a chunk of data, including a command that says "reload this data in 5 seconds", or "go load this other URL in 10 seconds".
- After the specified time has elapsed, the client either reloads the current data or gets the new data.



Server Push vs. Client Pull

- In server push, a HTTP connection is held open for an indefinite period of time (until the server sends a terminator, or until the client interrupts the connection).

- In client pull, a HTTP connection is never held open. Instead, the client is told when to open a new connection, and what data to get.

Client Pull Example: Periodical Reload

- A simple use of client pull is to periodically reload a document.

- For example, name the following document `reload1sec.html` and try loading it in web browser:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Refresh" CONTENT=1>
<title>Reload every second</title>
</HEAD>
<BODY>
<h1>This webpage is reloaded every second</h1>
</BODY>
</HTML>
```

- You will notice that the document reloads itself once a second.

Client Pull

- If we wanted to wait 10 seconds instead, we could have used this HTML command:

```
<META HTTP-EQUIV="Refresh" CONTENT=10>
```

- Make sure the META tag is inside the HEAD of your HTML document, before any displayed text or images.
- You can interrupt the infinite reloading by pressing the "Back" button.

Client Pull

- You can also cause another document to be reloaded in place of the current document.

- The META tag would be:

```
<META HTTP-EQUIV="Refresh" CONTENT="10;
  URL=http://ihome.ust.hk/~lixin/index.html">
```

- Important note: Use the full pathname in the URL (e.g. `http://whatever/whatever`). Do not use a relative URL.

Client Pull Example: Back and Forth

- The following example shows two HTML documents, `backforthA.html` and `backforthB.html`, each of which causes the other to load
  - If you load one, your browser will flip back and forth between them forever).

- Here is `backforthA.html`:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Refresh" CONTENT="1;
 URL=http://ihome.ust.hk/~lixin/backforthB.html">
<title>Back and Forth A</title>
</HEAD>
<BODY>
<h1>Back and forth part A.</h1>
</BODY>
</HTML>
```

- Here is `backforthB.html`:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Refresh" CONTENT="1;
 URL=http://ihome.ust.hk/~lixin/backforthA.html">
<title>Back and Forth B</title>
</HEAD>
<BODY>
<h1>Back and forth part B.</h1>
</BODY>
</HTML>
```

- When you load one of the documents, the browser will load the other in 5 seconds, then the first in another 5 seconds, and so on forever.

Server Push

- Server push is the other dynamic document method, complementing client pull.
- Unlike client pull, server push uses a connection that is held open over multiple responses, so the server can send more data any time it wants.
- The major advantage is that the server has total control over when and how often new data is sent.
- Also, this method can be more efficient, since new HTTP connections do not have to be opened all the time.
- Also, server push is easily interruptible (you can just hit "Stop" and interrupt the connection).

Accessing Server Push

- You should also store your server push CGI programs in `cgi-bin`.

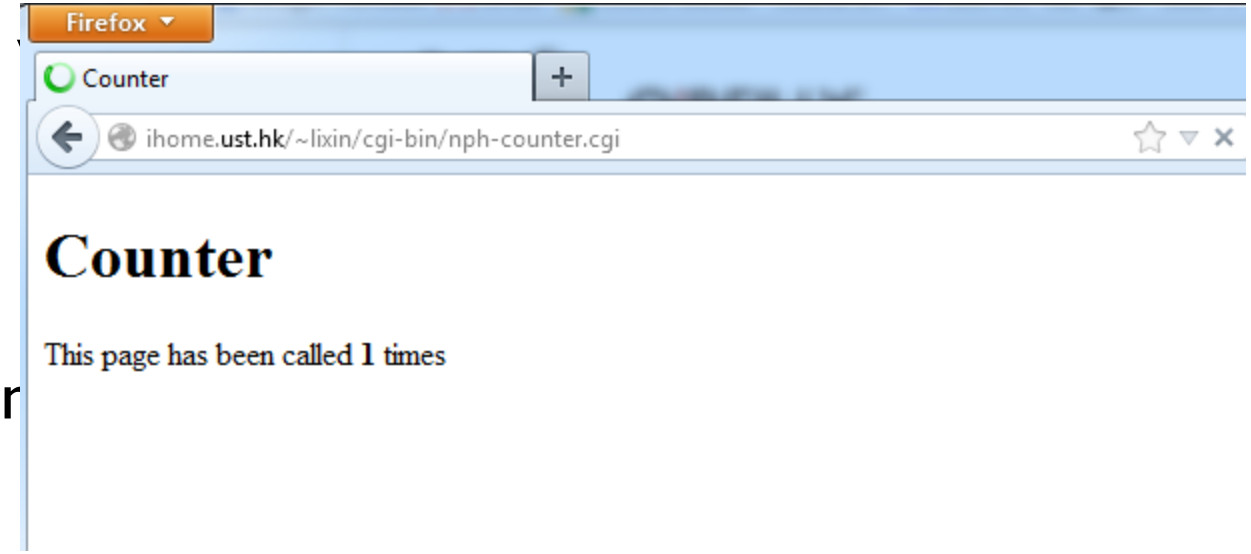
- Make sure that your filename starts with "nph-":

```
<P ALIGN=CENTER><B><font size="3" face="Arial">
<a href="http://ihome.ust.hk/~lixin/cgi-bin/nph-counter.cgi">
server push: counter</font></B></P>
```

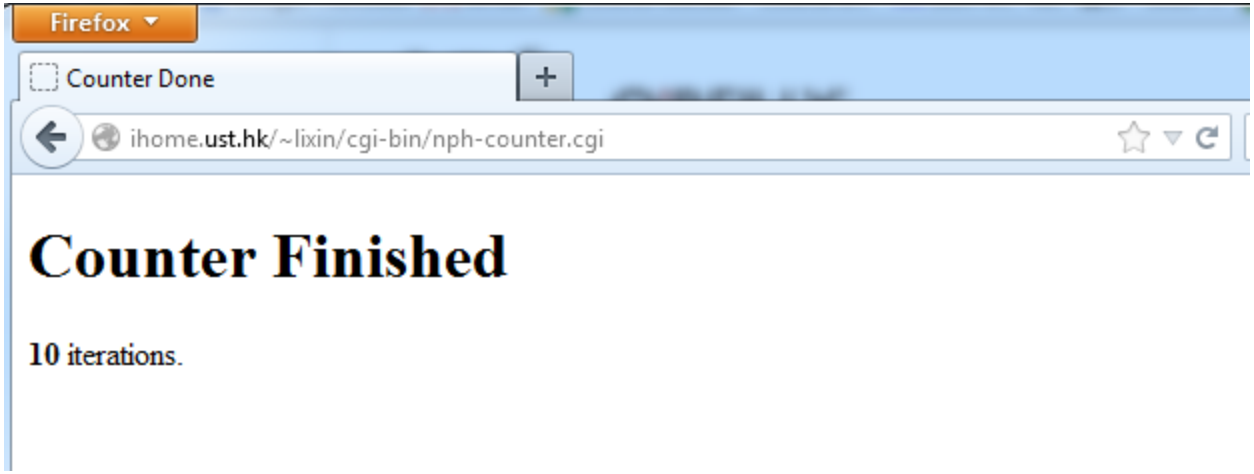
- The `nph` means "non-parsed headers" in the `cgiwrap` wrapper program. If you forget and use the regular `cgiwrap`, it will cause a "CGI Programming Error" message when you try to access the program.

Counter Example: Output

- The top screen
  10.



- Then the bottor



Counter Example page1

```
#!/usr/local/bin/perl
use CGI::Push qw(:standard);


do_push(-next_page=>\&next_page,
        -last_page=>\&last_page,
        -delay=>0.5);
```

Include CGI Push module

delay is number of seconds between reloads

`do_push()` takes as input references to 2 functions and a delay. The first function is called repeatedly until returning `undef`, when the last function is called once.

Counter Example page2

```
sub next_page {
 my($q,$counter) = @_;

 if($counter >= 10){
      return undef;
 }


 return start_html("Counter"),
      h1("Counter"),"\n",
      "This page has been called ", strong($counter),"
 times",
      end_html();
}
```

subroutine to generate HTML counter

The Push module will pass 2 scalars to the function. The second scalar is how many times the function has been called.

`return undef` to `break loop`

`Output the HTML to the return statement`

Counter Example page3

```
sub last_page {
  my($q,$counter) = @_;
  return start_html("Counter Done"),
       h1("Counter Finished"),
       strong($counter), " iterations.",
       end_html;
}
```

Same idea as next_page(), except this function is only called once at the end to display the final HTML page.

Date Example

- The following example shows the current time indefinitely (until the user hits the "Stop" or "Back" buttons), updating the time each second.

```
#!/usr/local/bin/perl
use CGI::Push qw(:standard);

do_push(-next_page=>\&date_page,
        -last_page=>\&date_page,
        -delay=>1.0)

sub date_page {
 return start_html("Date"),
      h1("Date"),"\n",
      "The current time is ",
 scalar(localtime),"\n",
      end_html();
}
```

same page used as last page

1-second delay

no exit condition, loops indefinitely

converts date output to easy-to-read format