CGI Scripts

CGI scripts can be written in *most* languages.
The better CGI languages:

- are good at manipulating character strings
- make it easy to produce HTML

Perl satisfies both of these criteria ok on its own.
Libraries like `CGI.pm` make Perl even better for CGI.

CGI at CSE

On CSE machines, users typically place CGI scripts in:
`/home/`*UserName*`/public_html/cgi-bin`
And access them via:
`http://cgi.cse.unsw.edu.au/~`*Username*`/cgi-bin/`*Script*
Nowadays, you can place CGI scripts

- anywhere under your `public_html` directory
- provided that they have a `.cgi` or suffix

and access them via e.g.
`http://cgi.cse.unsw.edu.au/~`*UserName*`/path/to/script.cgi`
The CSE web server will automatically forward them to the CGI server for execution.

CGI at CSE

A note on file/directory protections and security ...

- files under `public_html` need to be readable
- directories under `public_html` need to be executable

so that at least the Web server can access them.
A special command:

```
priv webonly FileOrDirecctory
```

makes files/dirs readable only to you and the web server.

CGI and Security

Putting up a CGI scripts means that

- anyone, anywhere can execute your script
- they can give it any data they like

If you are not careful how data is used ...
Many prople run Perl CGI scripts in "taint" mode

- generates an error if tainted data used unsafely

Tainted data = any CGI parameter
Unsafely = in system-type operations (e.g. ‘...‘)

CGI.pm

`CGI.pm` is a Perl module to simplify CGI scripts.
It provides functions/methods that make it easy

- to access parameters and other data for CGI scripts
- to produce HTML output from the script

CGI.pm supports two styles of programming:

- object-oriented, with CGI objects and methods
- function-oriented, with function calls (single implicit CGI object)

We'll use simpler function-oriented style in this course.

CGI.pm

CGI.pm has a range of methods/functions for:

- producing HTML (several flavours, including browser-specific),
- building HTML forms (overall wrapping, plus all form elements)
- CGI handling (manipulating parameters, managing state)

HTML and form building methods typically

- accept a collection of string arguments
- return a string that contains a fragment of HTML

A dynamic Web page is produced by

- printing a collection of such HTML fragments

Example CGI.pm

Consider a data collection form (`SayHello.html`):

```
<form name="Hello" action="HelloScript.cgi">
Your name: <input name="UserName" type="text">
<input type=submit value="Say Hello">
</form>
```

And consider that we type `John` into the input box.

Example CGI.pm

An OO-style script (`HelloScript.cgi`)

```
use CGI;
$cgi = new CGI;
$name = $cgi->param("UserName");
print $cgi->header(), $cgi->start_html(),
      $cgi->p("Hello there, $name"),
      $cgi->end_html();
```

Output of script (sent to browser):

```
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Untitled Document</TITLE>
</HEAD><BODY><P>Hello there, John</P></BODY></HTML>
```

Example CGI.pm

A function-style script (`HelloScript.cgi`)

```
use CGI qw/:standard/;
$name = param("UserName");
print header(), start_html(),
      p("Hello there, $name"),
      end_html();
```

Output of script (sent to browser):

```
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Untitled Document</TITLE>
</HEAD><BODY><P>Hello there, John</P></BODY></HTML>
```

Calling CGI.pm Methods

CGI.pm methods often accept many (optional) parameters.
Special method-call syntax available throughout CGI.pm:

```
MethodName(-ArgName1=>Value1,
           -ArgName2=>Value2,
           -ArgName3=>Value3,
           ...
           -ArgNamen=>Valuen,
           );
```

Example:

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

Argument names are case-insensitive; args can be supplied in any order.

Calling CGI.pm Methods

CGI.pm doesn't explicitly define methods for all HMTL tags.
Instead, constructs them on-the-fly using rules about arguments.
This allows you to include arbitrary attributes in HTML tags

```
MethodName(-AttrName=>Value,..., OtherArgs, ...);
```

If first argument is an associative array, it is converted into tag attributes.
Other unnamed string arguments are concatenated space-separated.
Methods that behave like this are called *HTML shortcuts*.

Calling CGI.pm Methods

Examples of HTML shortcuts:

`h1() or h1`	`<H1>`
`h1('some','contents')`	`<H1>some contents</H1>`
`h1({-align=>left})`	`<H1 ALIGN="left">`
`h1({-align=>left},'Head')`	`<H1 ALIGN="left">Head</H1>`
`p() or p`	`<P>`
`p('how's',"this","now")`	`<P>how's this now</P>`
`p({-align=>center},'Now!')`	`<P ALIGN="center">Now!</P>`

Accessing Data Items

The **param** method provides access to CGI parameters.

- can get a list of names for all parameters
- can get value for a single named parameter
- can modify the values of individual parameters

Examples:

```
# get a list of names of all parameters
@names = param();
# get value of parameter "name"
$name = param('name');
# get values of parameter "choices"
@list = param('choices');
# set value of "colour" parameter to 'red"
param('colour','red');
param(-name=>'colour', -value='red');
```

Accessing Data Items

Example - dump a table of CGI params:

```
#!/usr/bin/perl
use CGI ':standard';
@params = param();
print header, "<html><body>";

foreach $p (@params) {
    $v = param($p);
    $rows .= "<tr><td>$p</td><td>$v ";
}

print "<center><table border=1>
<tr><th>Param<th>Value
$rows
</table>
</body></center></html>";
```

Accessing Data Items

Example - dump a table of CGI params, using shortcuts:

```
#!/usr/bin/perl
use CGI ':standard';
@params = param();
print header, start_html;

foreach $p (@params) {
   $rows .= Tr(td([$p, param($p)]));
}
print center(
         table({-border=>1},
            Tr(th(['Param','Value'])),
            $rows
         )
      ),
      end_html;
```

Generating Forms

CGI.pm has methods to assist in generating forms dynamically:

`start_form`	generates a <form> tag with optional params for `action`,...
`end_form`	generates a </form> tag

Plus methods for each different kind of data collection element

- `textfield`, `textarea`, `password_field`
- `popup_menu`, `scrolling_list`
- `checkbox`, `radio_group`, `checkbox_group`
- `submit`, `reset`, `button`, `hidden`

Self-invoking Form

```
use CGI qw/:standard/;  # qw/X/ == 'X'
print header,
    start_html('A Simple Example'),
    h1(font({-color=>'blue'},'A Simple Example')),
    start_form,
    "What's your name? ",textfield('name'),p,
    "What's your favorite color? ",
    popup_menu(-name=>'color',
               -values=>['red','green','blue','yellow'])
    p,
    submit,
    end_form;
if (param()) {
    print "Your name is ",em(param('name')),p,
        "Your favorite color is ",em(param('color')),
        hr;
}
```

CGI Script Structure

CGI scripts *can* interleave computation and output.
Arbitrary interleaving is not generally effective
(e.g. produce some output and then encounter an error in middle of table)
Useful structure for (large) scripts:

- collect and check parameters; handle errors
- use parameters to compute result data structures
- convert results into HTML string
- output entire well-formed HTML string

Multi-page (State-based) Scripts

Often, a Web-based transaction goes through several stages.
Sometimes useful to implement all stages by a single script.
Such scripts are

- structured as a collection of cases, distinguished by a "state" variable
- each state sets parameter to pass to next invocation of same script
- new invocation produces a new state (different value of "state" variable)

Overall effect: a single script produces many different Web pages.

Multi-page (State-based) Scripts

Example (state-based script schema):

```
$state = param("state");
if ($state eq "") {
   do processing for initial state
   set up form to invoke next state
} elsif ($state == Value1) {
   do processing for state 1
   set up form to invoke next state
} elsif ($state == Value2) {
   do processing for state 2
   set up form to invoke next state
} elsif ($state == Value3) {
   do processing for state 3
   set up form to invoke next state
}
...
```

Cookies

Web applications often need to maintain state (variables) between execution of their CGI script(s).
Hidden input fields are useful for the one "session".
Cookies provide more persistant storage.
Cookies are strings sent to web clients in the response headers.
Clients (browsers) store these strings in a file and send them back in the header when they subsequently access the site. For example:

```
$ ./webget.pl  http://www.amazon.com/
HTTP/1.1 200 OK
Date: Thu, 19 May 2011 00:54:27 GMT
Server: Server
Set-Cookie: skin=noskin; path=/;domain=.amazon.com;expires=Thu, 19-May-2011 00:54:27 GMT
Set-cookie: session-id-time=2085672011;path=/;domain=.amazon.com;expires=Tue Jan 01 08:00:01 2036 GMT
Set-cookie: session-id=191-0575084-2685655;path=/;domain=.amazon.com;expires=Tue Jan 01 08:00:01 2036 GMT
```

Web clients send the cookie strings back next time they fetch pages from Amazon.

Storing a Hash

The Storable module provides an easy way to store a hash, e.g:

```
use Storable;
$cache_file = "./.cache";
%h = %{retrieve($cache_file)} if -r $cache_file;
$h{COUNT}++;
print "This script has now been run $h{COUNT} times\n";
store(\%h, $cache_file);
```

Source: http://cgi.cse.unsw.edu.au/~cs2041/code/web/persistent.pl

```
$ persistent.pl
This script has now been run 1 times
$ persistent.pl
This script has now been run 2 times
$ persistent.pl
This script has now been run 3 times
...
```

A Web Client with Cookies

We can add code to our simple web client to store cookies it receives using Storable.

```
use Storable;
$cookies_db = "./.cookies";
%cookies = %{retrieve($cookies_db)} if -r $cookies_db;
...
while (<$c>) {
  last if /^\s*$/;
  next if !/^Set-Cookie:/i;
  my ($name,$value, %v) = /([^=;\s]+)=([^=;\s]+)/g;
  my $domain = $v{'domain'} || $host;
  my $path = $v{'path'} || $path;
  $cookies{$domain}{$path}{$name} = $value;
  print "Received cookie $domain $path $name=$value\n"
}
```

Source: http://cgi.cse.unsw.edu.au/~cs2041/code/web/webget-cookies.pl

A Web Client with Cookies

And add code to send cookies when making requests:

```
use Storable;
$cookies_db = "./.cookies";
%cookies = %{retrieve($cookies_db)} if -r $cookies_db;
...
foreach $domain (keys %cookies) {
  next if $host !~ /$domain$/;
  foreach $cookie_path (keys %{$cookies{$domain}}) {
    next if $path !~ /^$cookie_path/;
    foreach $name (keys %{$cookies{$domain}{$path}}) {
      my $cookie = $cookies{$domain}{$path}{$name}
      print $c "Cookie: $cookie\n";
    }
  }
}
```

Source: http://cgi.cse.unsw.edu.au/~cs2041/code/web/webget-cookies.pl

A Web Client with Cookies

In action:

```
$ webget-cookies.pl http://www.amazon.com/
Received cookie .amazon.com / skin=noskin
Received cookie .amazon.com / session-id-time=2092797201l
Received cookie .amazon.com / session-id=192-8901109-68109
$ webget-cookies.pl http://www.amazon.com/
Sent cookie skin=noskin
Sent cookie session-id-time=2092797201l
Sent cookie session-id=192-8901109-6810988
Received cookie .amazon.com / skin=noskin
Received cookie .amazon.com / ubid-main=198-1199999-11869
Received cookie .amazon.com / session-id-time=2092797201l
Received cookie .amazon.com / session-id=192-8901109-68109
```

CGI Script Setting Cookie Directly

This crude script puts a cookie in the header directly.
And retrieves a cookie from the HTTP_COOKIE environment variable.

```
$x = 0;
if ($ENV{HTTP_COOKIE} =~ /\bx=(\d+)/) {
    $x = $1 + 1;
}
print "Content-type: text/html
Set-Cookie: x=$x;

<html><head></head><body>
x=$x
</body></html>";
```

Source: http://cgi.cse.unsw.edu.au/~cs2041/code/web/simple_cookie.cgi

Using CGI.pm to Set a Cookie

CGI.pm provides more convenient access to cookies.

```
use CGI qw/:all/;
use CGI::Cookie;

%cookies = fetch CGI::Cookie;
$x = 0;
$x = $cookies{'x'}->value if $cookies{'x'};
$x++;
print header(-cookie=>"x=$x");
print start_html('Cookie Example')
print "x=$x\n"
print end_html;
```

Source: http://cgi.cse.unsw.edu.au/~cs2041/code/web/simple_cookie.cgipm.cgi

CGI Security Vulnerability - Input Parameter length

CGI script may expect a parameter containing a few bytes, e.g. user name.
But a malicious user may supply instead megabytes.
This may the first step in a buffer overflow or denial of service attack
Always check/limit length of input parameters.

```
$user = param('user');
$user = substr $user, 0, 64;
```

CGI Security Vulnerability - Absolute Pathname and ..

CGI script may use a parameter has a filename.
A malicious user may supply an input containing / or ..
This will allow read and/or write access to other files on system.
Always santitize of input parameters.
Safest to remove all but necessary characters, e.g.:

```
$name = param('name');
$name = s/[^a-z0-9]//g;
```

CGI Security Vulnerability - Perl's Two Argument Open

The 2 argument version of Perl's open treats $>$ and — as special characters.
A malicious user may supply an input containing thgese characters
This will allow files to be written and arbitrary programs to be run.
Always santitize input parameters.
Safest to use 3 argument form of open.

CGI Security Vulnerability - User Input & External Programs.

A CGI script may pass user input as arguments to an external program. External program are often run via a shell, e.g. Perl's system and back quotes.
A malicious user may supply input containing shell metacharacters such as — or ;
This will allow arbitrary programs to be run.
Always santitize input parameters.
Safest to run external programs directly (not via shell).

CGI Security Vulnerability - SQL Injection

A CGI script may incorporate user input in SQL commands.
A malicious user may supply input containing SQL metacharacters such as '
This may allow the user to circumvent authentication.
Remove or quote SQL metacharacters before using them in queries.
Safest to run query via PREPARE.

CGI Security Vulnerability - Cross-site Scripting (XSS)

A CGI script may incorporate user input into web pages shown to other users.
A malicious user may supply input containing HTML particularly Javascript.
This Javascript can redirect links, steal information etc.
Remove $<$, $>$, & characters from input before incorporating in web pages.
In other contexts, e.g. within script tags, other characters unsafe.

Further Information ...

Comprehensive documentation attached to course Web page:
http://perldoc.perl.org/CGI.html
Most Perl books have some material on CGI.pm.