TCP/IP Intro

TCP/IP beyond scope of this course - take COMP[39]331.
But easier to understand CGI using TCP/IP from Perl
Easy to establish a TCP/IP connection.
Server running on host williams.cse.unsw.edu.au does this:

```
use IO::Socket;
$server = IO::Socket::INET->new(LocalPort => 1234,
                Listen => SOMAXCONN) or die;
$c = $server->accept()
```

Client running anywhere on internet does this:

```
use IO::Socket;
$host = "williams.cse.unsw.edu.au";
$c = IO::Socket::INET->new(PeerAddr=>$host,
                           PeerPort=>1234) or die;
```

Then $c effectively a bidirectional file handle.

Time Server

A simple TCP/IP server which supplies the current time as an ASCII string.

```
use IO::Socket;
$server = IO::Socket::INET->new(LocalPort => 4242,
                Listen => SOMAXCONN) or die;
while ($c = $server->accept()) {
    printf "[Connection from %s]\n", $c->peerhost;
    print $c scalar localtime,"\n";
    close $c;
}
```

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

Time Client

Simple client which gets the time from the server on host `$ARGV[0]` and prints it.
See NTP for how to seriously distribute time across networks.

```
use IO::Socket;
$server_host =  $ARGV[0] || 'localhost';
$server_port = 4242;
$c = IO::Socket::INET->new(PeerAddr => $server_host,
                           PeerPort  => $server_port) or die;
$time = <$c>;
close $c;
print "Time is $time\n";
```

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

Well-known TCP/IP ports

To connect via TCP/IP you need to know the port. Particular services often listen to a standard TCP/IP port on the machine they are running. For example:

- 21 ftp
- 22 ssh (Secure shell)
- 23 telnet
- 25 SMTP (e-mail)
- 80 HTTP (Hypertext Transfer Protocol)
- 123 NTP (Network Time Protocol)
- 443 HTTPS (Hypertext Transfer Protocol over SSL/TLS)

So web server normally listens to port 80 (http) or 443 (https).

Uniform Resource Locator (URL)

Familiar syntax:

```
scheme://domain:port/path?query_string#fragment_id
```

For example:

```
http://en.wikipedia.org/wiki/URI_scheme#Generic_syntax
http://www.google.com.au/search?q=COMP2041&hl=en
```

Given a http URL a web browser extracts the hostname from the URL and connects to port 80 (unless another port is specified).
It then sends the remainder of the URL to the server.
The HTTP syntax of such a request is simple:

`GET` *path* `HTTP/`*version*

We can do this easily in Perl

Simple Web Client in Perl

A very simple web client - doesn't render the HTML, no GUI, no ... - see HTTP::Request::Common for a more general solution

```
use IO::Socket;
foreach $url (@ARGV) {
    $url =~ /http:\/\/([^\/]+)(:(\d+))?(.*)/ or die;
    $c = IO::Socket::INET->new(PeerAddr => $1,
            PeerPort => $2 || 80) or die;
    # send request for web page to server
    print $c "GET $4 HTTP/1.0\n\n";
    # read what the server returns
    my @webpage = <$c>;
    close $c;
    print "GET $url =>\n", @webpage, "\n";
}
```

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

Simple Web Client in Perl

```
$ cd /home/cs2041/public_html/lec/cgi/examples
$ ./webget.pl http://cgi.cse.unsw.edu.au/
GET http://cgi.cse.unsw.edu.au/ =>
HTTP/1.1 200 OK
Date: Sun, 21 Sep 2014 23:40:41 GMT
Set-Cookie: JSESSIONID=CF09BE9CADA20036D93F39B04329DB
Last-Modified: Sun, 21 Sep 2014 23:40:41 GMT
Content-Type: text/html;charset=UTF-8
Content-Length: 35811
Connection: close

<!DOCTYPE html>
<html lang='en'>
  <head>
...
```

Notice the web server returns some header lines and then data.

Web server in Perl - getting started

This Perl web server just prints details of incoming requests & always returns a 404 (not found) status.

```
use IO::Socket;
$server = IO::Socket::INET->new(LocalPort => 2041,
          ReuseAddr => 1, Listen => SOMAXCONN) or die;
while ($c = $server->accept()) {
    printf "HTTP request from %s =>\n\n", $c->peerhost;
    while ($request_line = <$c>) {
        print "$request_line";
        last if $request_line !~ /\S/;
    }
    print $c "HTTP/1.0 404 This server always 404s\n";
    close $c;
}
```

http://cgi.cse.unsw.edu.au/ cs2041/code/web/webserver-404.pl

Web server in Perl - getting started

```
use IO::Socket;
$server = IO::Socket::INET->new(LocalPort => 2041,
          ReuseAddr => 1, Listen => SOMAXCONN) or die;
$content = "Everything is OK - you love COMP[29]041.\n";
while ($c = $server->accept()) {
    printf "HTTP request from %s =>\n\n", $c->peerhost;
    while ($request_line = <$c>) {
        print "$request_line";
        last if $request_line !~ /\S/;
    }
    my $request = <$c>;
    print "Connection from ", $c->peerhost, ": $request";
    $request =~ /^GET (.+) HTTP\/1.[01]\s*$/;
    print "Sending back /home/cs2041/public_html/$1\n";
    open my $f, '<', "/home/cs2041/public_html/$1";
    $content = join "", <$f>;
    print $c "HTTP/1.0 200 OK\nContent-Type: text/html\n";
    print $c "Content-Length: ",length($content),"\n";
    print $c $content;
    close $c;
}
```

Web server in Perl - too simple

A simple web server in Perl.
Does fundamental job of serving web pages but has bugs, securtity holes and huge limitations.

```
while ($c = $server->accept()) {
    my $request = <$c>;
    $request =~ /^GET (.+) HTTP\/1.[01]\s*$/;
    open F, "</home/cs2041/public_html/$1";
    $content = join "", <F>;
    print $c "HTTP/1.0 200 OK\n";
    print $c "Content-Type: text/html\n";
    print $c "Content-Length: ",length($content),"\n";
    print $c $content;
    close $c;
}
```

Source: http://cgi.cse.unsw.edu.au/~cs2041/code/web/webserver-too-simple.pl

Web server in Perl - mime-types

Web servers typically determine a file's type from its extension (suffix) and pass this back in a header line.
ON Unix-like systems file /etc/mime.types contains lines mapping extensions to mime-types, e.g.:

```
application/pdf                 pdf
image/jpeg                      jpeg jpg jpe
text/html                       html htm shtml
```

May also be configured within web-server e.g cs2041's .htaccess file contains:

```
AddType text/plain pl py sh c cgi
```

Web server in Perl - mime-types

Easy to read /etc/mime.types specifications into a hash:

```
open MT, '<', "/etc/mime.types") or die;
while ($line = <MT>) {
    $line =~ s/#.*//;
    my ($mime_type, @extensions) = split /\s+/, $line;
    foreach $extension (@extensions) {
     $mime_type{$extension} = $mime_type;
    }
}
```

Web server in Perl - mime-types

Previous simple web server with code added to use the `mime_type` hash to return the appropriate `Content-type`:

```
$url =~ s/(^|\/)\.\.(\/|$)//g;
my $file = "/home/cs2041/public_html/$url";
# prevent access outside 2041 directory
$file =~ s/(^|\/)..(\/|$)//g;
$file .= "/index.html" if -d $file;
if (open my $f, '<', $file) {
    my ($extension) = $file =~ /\.(\w+)$/;
    print $c "HTTP/1.0 200 OK\n";
    if ($extension && $mime_type{$extension}) {
        print $c "Content-Type: $mime_type{$extension}\n";
    }
    print $c <my $f>;
}
```

Source: http://cgi.cse.unsw.edu.au/~cs2041/code/web/webserver-mime-types.pl

Web server in Perl - multi-processing

Previous web server scripts serve only one request at a time.
They can not handle a high volume of requests.
And slow client can deny access for others to the web server, e.g our previous web client with a 1 hour sleep added:

```
$url =~ /http:\/\/([^\/]+)(:(\d+))?(.*)/ or die;
$c = IO::Socket::INET->new(PeerAddr => $1,
      PeerPort => $2 || 80) or die;
sleep 3600;
print $c "GET $4 HTTP/1.0\n\n";
```

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

Simple solution is to process each request in a separate process.
The Perl subroutine fork duplicates a running program.
Returns 0 in new process (child) and process id of child in original process (parent).

Web server in Perl - multi-processing

We can add this easily to our previous webserver:

```
while ($c = $server->accept()) {
    if (fork() != 0) {
        # parent process goes to waiting for next request
        close($c);
        next;
    }
    # child processes request
    my $request = <$c>;
    ...
    close $c;
    # child must terminate here otherwise
    # it would compete with parent for requests
    exit 0;
}
```

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

Web server - Simple CGI

Web servers allow dynamic content to be generated via CGI (and other ways).
Typically they can be configure to execute programs for certain URIS.
for example cs2041's .htaccess file indicates files with suffix `.cgi` should be executed.

```
<Files *.cgi>
SetHandler application/x-setuid-cgi
</Files>
```

Web server - Simple CGI

We can add this to our simple web-server:

```
if ($url =~ /^(.*\.cgi)(\?(.*))?$/) {
    my $cgi_script = "/home/cs2041/public_html/$1";
    $ENV{SCRIPT_URI} = $1;
    $ENV{QUERY_STRING} = $3 || '';
    $ENV{REQUEST_METHOD} = "GET";
    $ENV{REQUEST_URI} = $url;
    print $c "HTTP/1.0 200 OK\n";
    print $c `$cgi_script` if -x $cgi_script;
    close F;
}
```

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

Web server - CGI

A fuller CGI implementation implementing both GET and POST requests can be found here:

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

HTML & CSS

- We're (hopefully) all familiar with HTML .
- HTML & CSS not covered (much) in lectures.
- If not familiar with HTML & CSS, may need to do extra reading.
- Tutes & labs will help.

Semantic Markup

- Use HTML tags to indicate the nature of the content.
- Use CSS to indicate how content type should be displayed

Document Object Model

- content marked up with *tags* to describe appearance
- browser reads HTML and builds Document Object Model (DOM)
- browser produces a visible rendering of DOM

HTML Document

```
<html>
<head><title>...</ti
<body bgcolor=white>
<h1>My Page</h1>
The first paragraph
contains lots of
<em>really</em>
interesting stuff.
<p>
But the second parag
is a bit boring.
</body>
</html>
```

Rendering by browser

My Page

The first parapgaph contains lots of *really* interesting stuff.

But the second paragraph is a bit boring.

Dynamic Web Pages

HTML tags are *static* (i.e. produce a fixed rendering).
"Dynamic" web content can be generated on server

- Generated on the server:
  - SSP (program running in web server generates HTML)
  - CGI (program running outside web server generates HTML)
  - many other variants
- Generated in the browser
  - JavaScript (browser manipulates document object model)
  - Java Applet (JVM in browser executes Java program) - dead
  - Silverlight (Microsoft browser plugin) - dying
  - Flash (Adobe browser plugin) - dying

Dynamic Web Pages

For CGI and SSP, the scripts (HTML generators) are invoked

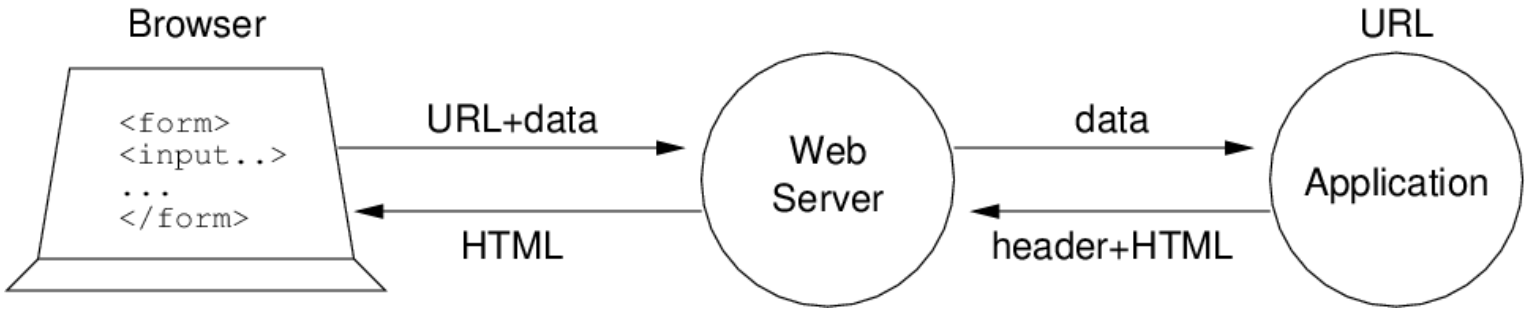
- via a URL (giving the name and type of application)
- passing some data values (either in the URL or via stdin)

The data values are typically

- collected in a fill-in form which invokes the script
- passed from page to page (script to script) via GET/POST

(other mechanisms for data transmission include cookies and server-state)

CGI (Common Gateway Interface)



Data is passed as `name=value` pairs (all values are strings).
Application outputs (normally) HTML, which server passes to client.
For HTML documents, header is `Content-type: text/html`
Header can be any MIME-type (e.g. `text/html`, `image/gif`, ...)

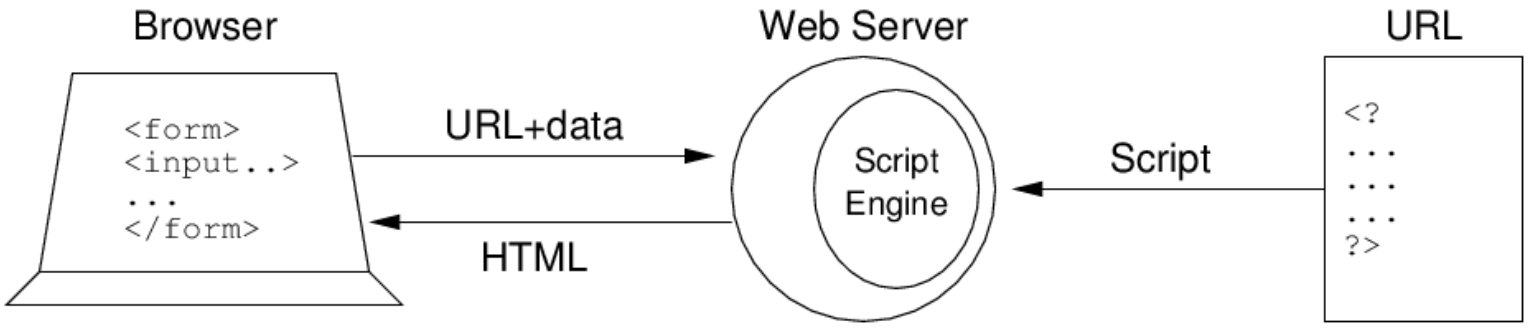
Perl and CGI

So how does Perl fit into this scenario?
CGI scripts typically:

- do lots of complex string manipulation
- write many complex strings (HTML) to output

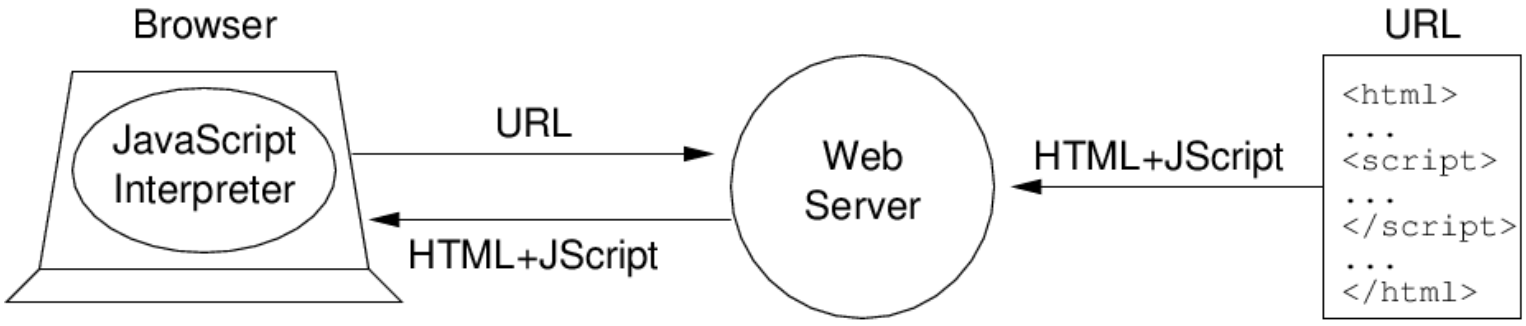
Perl is good at manipulating strings - good for CGI.
Libraries for Perl make CGI processing even easier.
CGI.pm is one such library (see later for more details)

SSP (Server-side Programming)



Data is available via library functions (e.g. `param`).
Script produces HTML output, which is sent to client (browser).

JavaScript (Client-side DOM Scripting)



Executing script can modify browser's internal representation of document (DOM)
Browser then changes appearance of document on screen.
This can happen at script load time or in response to *events* (such as `onClick`, `onMouseOver`, `onKeyPress`) after script has loaded.
Can also access data in form controls (because they are also document elements).

JavaScript (Client-side DOM Scripting)

For example, this web page has JavaScript embedded to sum two numbers from input fields and store the result in a third field.
The function is run whenever a character is entered in either field.

```
<input type=text id="x" onkeyup="sum();"> +
<input type=text id="y" onkeyup="sum();"> =
<input type=text id="sum" readonly="readonly">
<script type="text/javascript">
function sum() {
  var x = parseInt(document.getElementById('x').value);
  var y = parseInt(document.getElementById('y').value);
  document.getElementById('sum').value = num1 + num2;
}
</script>
```

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

Ajax

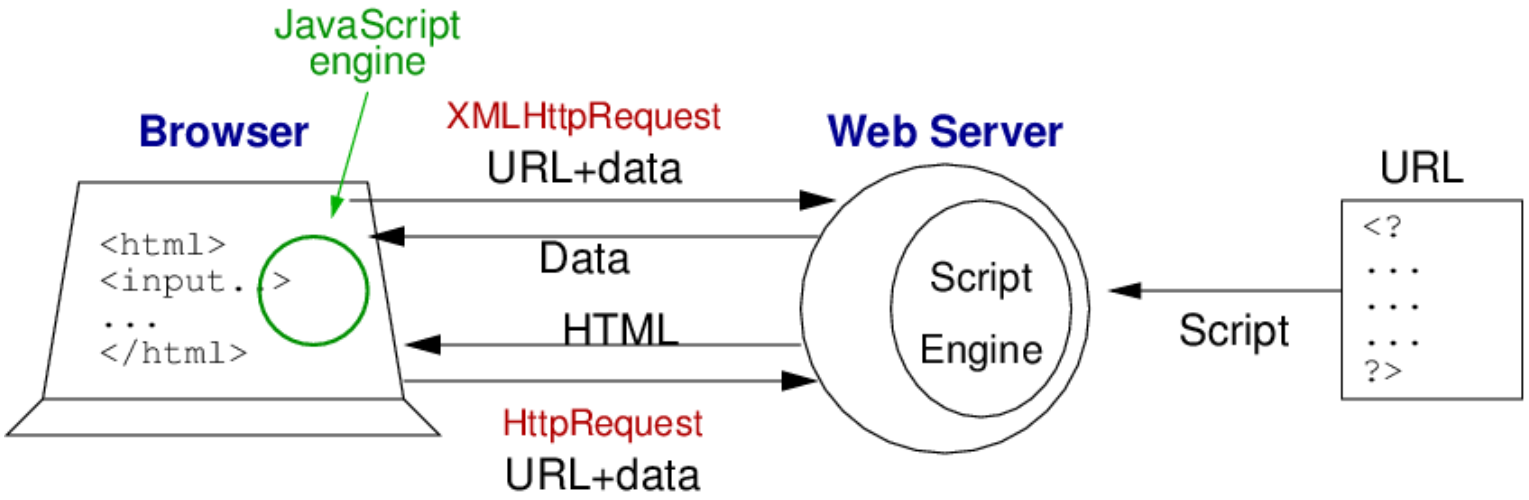
Ajax provides a variation on the above approach:

- "normal" browser-to-server interaction is HTTP request
- this causes browser to read response as HTML (new page)
- Ajax sends XMLHttpRequests from browser-to-server
- browser does not refresh, but waits for a response
- response data (not HTML) is read and added into DOM

Leads to interaction appearing more like traditional GUI.
Examples: Gmail, Google calendar, Flickr, ...
The popular JQuery library is an easy way to use AJAX.

Ajax

Ajax-style browser/server interaction:



Ajax showing result of matching Perl regex

```
$(document).ready(
 function() {
  $("#match").click(
   function() {
    $.get(
     "match.cgi",
     {string:$("#string").val(), regex:$("#regex").val()},
     function(data) {
      $("#show").html(data)
     }
    )
   }
  )
 }
)
```

Source: http://cgi.cse.unsw.edu.au/~cs2041cgi/code/web/match.html

Ajax

A new page is not loaded when the match button is pressed.
JQuery only updates a field on the page.
It fetches by http the results of the match from this CGI script:

```
use CGI qw/:all/;
print header;
if (param('string') =~ param('regex')) {
    print b('Match succeeded, this substring matched: ');
    print tt(escapeHTML($&));
} else {
    print b('Match failed');
}
```

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

HTML Forms

An HTML *form* combines the notions of *user input* & *function call* :

- collects data via *form control* elements
- invokes a URL to process the collected data when `submitted`

Syntax:

```
<form method=RequestMethod action=URL ...>
any HTML except another form
   mixed with
data collection (form control) elements
</form>
```

An HTML document may contain any number of `<form>`'s. Forms can be arbitrarily interleaved with HMTL layout elements (e.g. `<table>`)

`METHOD` Attribute

The *RequestMethod* value indicates how data is passed to `action` URL.
Two *RequestMethod*s are available: `GET` and `POST`

- `GET`: data attached to URL
  (*URL*?*name_1*=*val_1*&*name_2*=*val_2*&...)
- `POST`: data available to script via standard input

Within a server script all we see is a collection of variables:

- with the same names as those used in the form elements
- initialised with the values collected in the form

URL-encoded Strings

Data is passed from browser to server as a single string in the form:
name=*val*&*name*=*val*&*name*=*val*&...
with no spaces and where '=' and '&' are treated as special characters.
To achieve this strings are "url-encoded" e.g:

andrewt	andrewt
John Shepherd	John+Shepherd
~cs2041 = /home/cs2041	%7Ecs2041+%3D+%2Fhome%2Fcs2041
1 + 1 = 2	1+%2B+1+%3D+2
Jack & Jill = Love!	Jack+%26+Jill+%3D+Love%21

URL-encoded strings are usually decoded by library before your code sees them.

ACTION Attribute

<form ... **action**='*URL*' ... >

- specifies script *URL* to process form data

When the form is submitted ...

- invoke the URL specified in `action`
- pass all form data to it

If no `action` attribute, re-invoke the current script.

Other <form> Attributes

<form ... **name**='*FormName*' ... >

- associates the name *FormName* with the entire form
- useful for referring to form in JavaScript

<form ... **target**='*WindowName*' ... >

- causes output from executing script to be placed in specified window
- useful when dealing with frames (see later)

<form ... **onSubmit**='*Script*' ... >

- specifies actions to be carried out just before sending data to script

Form Controls

Form controls are the individual data collection elements within a form.
Data can be collected in the following styles:

text	single line or region of text
password	single line of text, value is hidden
menu	choose 1 or many from a number of options
checkbox	on/off toggle switch
radio	choose only 1 from a number of options
hidden	data item not displayed to user
submit	button, sends collected data to script
reset	button, resets all data elements in form
button	button, effect needs to be scripted

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.