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.

```perl
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.

```perl
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:

```perl
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:

```perl
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

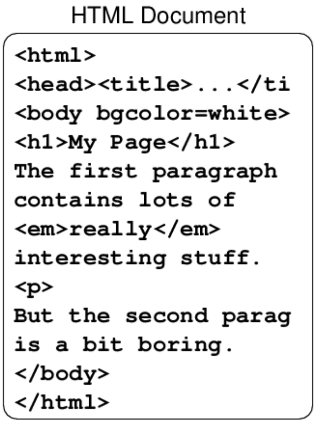
- We're (hopefully) all familiar with HTML .
- HTML & CSS not covered (much) in lectures.
- If not familiar with HTML & CSS, may need to do extra reading.
- Tutes & labs will help.

Semantic Markup

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

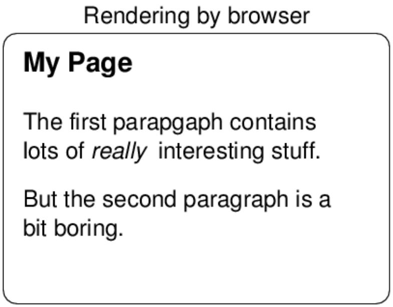
Document Object Model

- 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

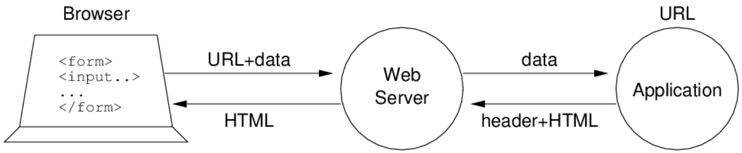
- via a URL (giving the name and type of application)
- passing some data values (either in the URL or via stdin)

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`, ...)

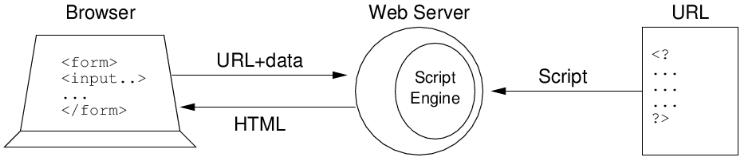
Perl and CGI

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

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

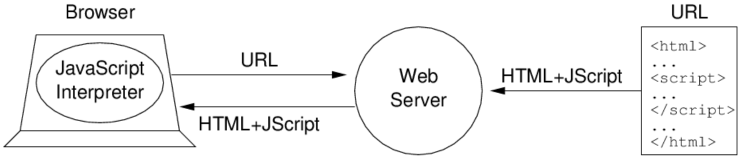
Perl is good at manipulating strings - good for CGI.
Libraries for Perl make CGI processing even easier.
CGI.pm is one such library (see later for more details)

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

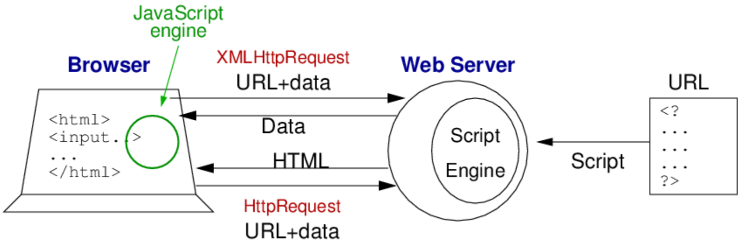
Ajax provides a variation on the above approach:

- “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