



Lesson #5

CGI/Perl and HTML Forms



About Perl

- Perl stands for *Practical Extraction and Report Language*.
- Created in 1986 by Larry Wall.
- Used to make web pages interactive.
- Very powerful text manipulation tool.
- Easily moved from one platform to another.



About CGI

- CGI stands for *Common Gateway Interface*.
- CGI is a protocol, not a programming language. It determines how servers talk to programs.
- CGI scripts are not all written in Perl, some are written in C, TCL, Ruby, PHP, Python, or VB.
- The CGI/Perl combination is the first and one of the most stable.



Perl and HTML

- Perl programs are executed on the server to generate dynamic HTML code.
- Specific Perl instructions are used to generate the code.
- A Perl application reacts with data submitted by the user to create a different page accordingly.



Perl Data

- Data in Perl can be classified as number or string.
- Data in Perl can be a constant or a variable.
- Data in Perl can be a scalar, an array or a hash.
- Numbers are the simplest. An expression in Perl is just represented as it is: $2 + 2$.



Numbers and Strings

- A string is a collection of characters.
- Strings are enclosed between " or '.
- Strings can be concatenated using the . (dot) operator. 'ice'.'cream' results in 'icecream'.
- Perl automatically assume the correct operands: $3 + 'a'$ is 3 and $3.'a'$ is '3a'.



Constants and Variables

- Like in all programming languages, variables are containers of values.
- Variable names always start with a sigil (\$, @ or %) depending if the variable is a scalar, an array or a hash.
- Variables do not need to be declared and are never typed.

```
$address = '3 Main Street';  
$x = $x + 10;
```




Scalars and Arrays

- A scalar is an individual piece of data. 'hello' and 3 are scalar constants. A scalar variable always begin with a \$.

```
$x = 10;
```

- An array consist of multiple values. An array in Perl can contain numbers and strings.

```
@array = (10, 'Toyota', 'Prius', 2015);  
print $array[1];  
print @array;
```

→ Toyota

→ 10ToyotaPrius2015



Hashes

- A hash is made up of pairs of scalars. The first element is called the key, the second element is the value. Keys and values are separated by `=>`. Pairs are separated by a comma `,`.

```
%info = (age=>45, species=>'lion');
```

- You can also use commas only (use quotes for keys).

```
%info2 = ('age',45,'species','lion');
```



Operators and Functions

- Basic operators are `+`, `-`, `*`, `/`, `%` and `.`

```
$sum = 3 + 5;
```

```
$modulus = 3 % 5;
```

```
$quotient = 3 / 5;
```

- Functions have their arguments enclosed in parentheses.

```
$item1 = shift (@array);
```



Operators and Functions

- The result of a function is usually different than its return value. For `shift ('a','b','c')`, the **result** is `('b','c')`, the **return value** is `'a'`. (*shift* removes the first element of the array)

```
@array1 = ('a','b','c');  
$array2 = shift (@array1);  
print @array1;  
print $array2;
```

bca



Quotation Marks

- There are 2 kind of quotation marks in Perl.
- The single quote (') is used when you want the string to be taken literally.

```
$word = "World";  
print 'Hello\n$word';  
Hello\n$word
```

- The double quote (") is used when you want the string to be evaluated.

```
$word = "World";  
print "Hello\n$word";  
Hello World
```

← New lines are ignored
in HTML!



Quotation Functions

- There are 3 functions, one for each kind of quotations marks. Makes it easier.
- **q** for a single quote, **qq** for double quotes and **qw** for quoting individual words.

```
$a1 = q(winner's circle);
```

```
$a2 = 'winner\'s circle';
```

```
$b1 = qq();
```

```
$b2 = "<img src=\"lamp.gif\">";
```

```
@c1 = qw(cats dogs elephants);
```

```
@c2 = ('cats','dogs','elephants');
```



Perl Statements

- As you may have already noticed, Perl statements are very similar to C statements.
- Like in C, the semicolon indicates a statement end.
- Also like C, curly braces { } are used to group blocks of statements.
- Unlike C, however, variables do not need to be declared (unless you need private variables).



Private Variables

- By default, all variables are global, they are visible through the entire program.
- It is possible to declare private variables, variables that are visible only inside their current block { }. These private variables declarations must occur inside the intended block. The **my** function declares private variables. Private variables are highly recommended if you borrow blocks of code not written by you.

```
my $car;  
my ($make, $model, $year);
```




The Shebang Line

- Equivalent to C preprocessor directives, the shebang line (sharp(#) and bang(!)), indicates the location of the Perl interpreter. The location is usually given to you by your hosting provider.
- Typical UNIX shebang line (mandatory):
`#!/usr/bin/perl`
- Typical Windows shebang line (optional):
`#!C:\Perl\bin\perl.exe`



Program Skeleton

- Programs are written using a simple text editor. Mandatory statements are in red, optional statements in blue.

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

```
use CGI ':standard';
```

```
use strict;
```

```
print "Content-type: text/html\n\n";
```

```
rest of program here...
```

- All you need after that is to save your file with a .cgi or .pl extension.



Program Hosting

- Always name your files with all-lower-case names. Underscores are OK. No spaces in file names.
- Upload the files to your cgi-bin directory on your host server (usually inside your public_html or www folder). Make sure that the directory protection is set at 711 or 755.
- Make sure your .cgi or .pl file has a protection of 711 or 755.



Getting Data

- A dynamic web page created by Perl becomes truly dynamic by processing information submitted by visitors.
- The easiest way is to submit information via HTML forms.

```
<form  
  action="http://www2.scs.ryerson.ca/~  
  dhamelin/cgi-bin/form1.cgi"  
  method="get">  
  First name: <input type="text"  
  name="first"><br>  
  Last name: <input type="text"  
  name="last"><br>  
  <input type="submit">  
</form>
```



The `<form>` Tag

- The form element creates a form for user input. A form can contain text fields, check boxes, radio buttons and more. Forms are used to pass user data to a specified URL.
- ***action*** is a required attribute. Its value is a URL that defines where to send the data when the submit button is pushed. That URL points to a server-side (Perl) program.
- ***method*** and ***name*** are also commonly used attributes.



The Method Attribute

- The HTTP method for sending data to the action URL. The default method is **get**.
- **method="get"**: This method sends the form contents in the URL: `URL?name=value&name=value`. Note: If the form values contains non-ASCII characters or exceeds 100 characters you **MUST** use **method="post"**.
- **method="post"**: This method sends the form contents in the body of the request. Note: Most browsers are unable to bookmark post requests.



The `<input>` Tag

- The `<input>` tag defines the start of an input field where the user can enter data.
- There are different types of input fields which are specified by the type attribute.
- **`type="text"`**: The default attribute. A simple text field.
- **`type="password"`**: A text field with invisible characters. Not recommended with get method!
- **`type="submit"`**: A button that will submit the form to the program.



The `<input>` Tag

- **`type="reset"`**: Brings back a blank form.
- **`type="checkbox"`**: With check boxes, more than one option can be selected.
- **`type="radio"`**: With radio buttons, only one option can be selected.

```
<form action="cgi-bin/test.cgi">  
Name:<input type="text" name="name"  
style="width:250px;">  
</form>
```

Name:



Password, Submit and Reset

```
<form action="cgi-bin/test.cgi">
```

```
Password: <input type="password"
name="password" style="width:200px">
<br>
```

```
<input type="submit">
```

```
<input type="reset">
```

```
</form>
```

Password:



Check Boxes

```
<form action="cgi-bin/test.cgi">  
Form of transportation?<br />  
Bike: <input type="checkbox" name="Bike">  
Car: <input type="checkbox" name="Car"><br>  
Foot: <input type="checkbox" name="Foot">  
Subway: <input type="checkbox" name="Subway">  
</form>
```

The data is in the name!

Form of transportation?
Bike: ☐ Car: ☐
Foot: ☒ Subway: ☒



Radio Buttons

```
<form action="city.cgi">  
City?<br>  
Toronto: <input type="radio"  
        name="city" value="T">  
Montreal: <input type="radio"  
          name="city" value="M">  
Vancouver: <input type="radio"  
            name="city" value="V">  
</form>
```

City?

Toronto: ☐ Montreal: ☐ Vancouver: ☐



value and checked

- The **value** attribute just places a default value in a text field or assigns a value to a check box or radio button item.
- The **checked** attribute will check a radio button or check box by default.

```
<form action="test.cgi">
```

```
Section?<br>
```

```
011: <input type="checkbox" name="section"
       value="011" checked="checked">
```

```
021: <input type="checkbox" name="section"
       value="021">
```

```
</form>
```

The data
is
in the
value!

Section?

011: ☒ 021: ☐



The `<textarea>` Tag

- Defines a text-area (a multi-line text input control). A user can write text in the text-area. In a text-area you can write an unlimited number of characters. The default font in the text-area is fixed pitch.
- The required attributes are rows and cols for the numbers or rows and columns visible.
- Other attributes are name, and readonly.



The <textarea> tag

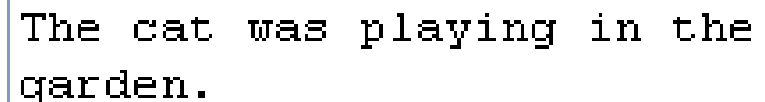
```
<form action="cgi-bin/test.cgi">
```

```
<textarea rows="10" cols="30"  
  readonly="readonly">
```

```
The cat was playing in the garden.
```

```
</textarea>
```

```
</form>
```

A screenshot of a web browser window. It shows a single text area containing the text "The cat was playing in the garden." The text area has a blue border and a vertical scrollbar on the right side. The text is in a monospaced font.



Menus and Combo Boxes

- Menus and combo boxes are created with the `<select>` and `<option>` tags.
- Attributes include name, size (the number of visible elements) and `multiple="multiple"` (allowing multiple items to be selected).

```
<select name="car" size="2" multiple="multiple">  
  <option value="volvo">Volvo</option>  
  <option value="saab">Saab</option>  
  <option value="opel">Opel</option>  
  <option value="audi">Audi</option>  
</select>
```





Submit and Reset Buttons

- By default, the submit button label is *Submit Query*. You can change it with the value attribute.
- By default, the reset button label is *Reset*. You can change it with the value attribute.

```
<input type="submit" value="Go!">
```

```
<input type="reset" value="Erase">
```

A rectangular button with a blue border and a light gray background, containing the text "Go!" in a monospace font.

A rectangular button with a blue border and a light gray background, containing the text "Erase" in a monospace font.



Using a Query String

- An alternative to forms exist to submit data to a CGI/Perl program. We can use the query string.
- A query string item consists of a key and a value. The key corresponds to the name of a form item and the value, its value.
- See a script in action here:

www2.scs.ryerson.ca/~dhamelin/cgi-bin/test.cgi?city=Toronto&day=Monday



Environment Variables

- Environment variables are set each time a *CGI* script is run.
- They contain information about the Web server, its configuration and about the request made by the browser.
- Environment variables vary from server to server. They are stored in the `%ENV` hash.



HTTP_USER_AGENT

- The HTTP_USER_AGENT contains the name and version of the visitor's browser, the computer platform.

```
print $ENV{'HTTP_USER_AGENT'};
```

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)

Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7)

Gecko/20040803 Firefox/0.9.3



Viewing Environment Variables

- Here a simple program that shows all the available environment variables on a server.

```
foreach (keys %ENV) {  
    print "Key: $_; Value:  
$ENV{$_}";  
}
```

- See it in action here:

www2.scs.ryerson.ca/~dhamelin/cgi-bin/env.cgi

ihypres.net/programming/perl/exec.php?script=02



Getting Single-Valued Form Data

- The form (saved in public_html directory):

```
<form action="cgi-bin/name.cgi" method="post">
```

```
Name: <input type="text"
      name="name"
      style="width:300px;">
```

```
<input type="submit">
```

```
</form>
```




Getting Single-Valued Form Data

- The name.cgi script (saved in cgi-bin directory with 755 protection):

```
#!/usr/bin/perl
use CGI ':standard';
print "Content-type: text/html\n\n";

$name = param ('name');
print "Hello $name";
```

See it in action here:

www.scs.ryerson.ca/dhamelin/courses/cps530/name1.html



Getting Multiple-Valued Form Data

- The form (saved in public_html directory):

```
<form action="cgi-bin/ckbox.cgi"
  method="post">
```

Select your favorite dish (you may pick more than one):


```
<input type="checkbox" name="dish"
  value="pizza">Pizza
```

```
<input type="checkbox" name="dish"
  value="tandoori">Tandoori Chicken
```

```
<input type="checkbox" name="dish"
  value="steak">Steak <br>
```

```
<input type="submit">
```

```
</form>
```



Getting Multiple-Valued Form Data

- The ckbox.cgi script (saved in cgi-bin directory with 755 protection):

```
#!/usr/bin/perl  
use CGI ':standard';  
print "Content-type:  
text/html\n\n";
```

```
@food = param ('dish');  
print "Your favorite dishes are:  
@food";
```

See it in action here:

www.scs.ryerson.ca/dhamelin/courses/cps530/food.html



Getting the Form's Names

- The `param` function without arguments supplies an array containing the form items' names.

```
@names = param();
```

- To get the values as well, use a `foreach` loop.

```
foreach my $name (param()) {  
    my @values = param($name);
```



Scalar Operators

- `::` assignment operator.
- `+`, `-`, `*`, `/`, `%` are like in C.
- Operator precedence rules are similar to those in C.
- `**` is the exponent operator. It takes precedence over `*`, `/` and `%`.
- Dot (`.`) is the string concatenation operator.
- The math functions *sqrt*, *abs*, *int*, *atan2*, *cos*, *sin*, *log* and *exp* are available.
- Shortcuts like `$a+=3;` and `++$i` are permitted.



Comparison Operators

- `<`, `<=`, `>`, `>=`, `==` and `!=` are used to compare numbers.
- Strings use a different set of operators:
- `eq` for equal, `ne` for not equal, `gt` for greater than, `ge` for greater or equal, `lt` for less than and `le` for less or equal.
- A condition is true if not equal to zero, empty or undefined.



if statement

- Some examples of if statements:

```
if ($food eq "spinach")
{
    print "Good! You will be
strong and healthy!";
}
else
{
    print "You should eat your
spinach!";
}
```




if statement

```
if ($food eq "spinach") {  
    print "Good! You will be  
    strong and healthy!";  
}  
→ elseif ($food eq "broccoli"){  
    print "Excellent! Broccoli  
    is good for you!";  
}  
else {  
    print "You should eat more  
    vegetables!";  
}
```



unless statement

- The unless statement is used for a false condition to simplify a program.

```
unless ($food eq "spinach")  
{  
    print "You should eat your  
    spinach!";  
}
```



foreach Statement

- The foreach statement is used to loop through all the elements of an array.

```
foreach $creature (@prey)
{
    print "<div>$creature</div>";
}
```

- In the previous code, we take the first element of the @prey array, assign it to the \$creature variable. We do that for all the elements of the array in turn.



while Statement

- The while statement repeats a block while the condition remains true.

```
$i = 1;  
while ($i <= 100) {  
    print "$i ";  
    ++$i;  
}
```



until Statement

- The *until* statement repeats a block while the condition remains false or until it is true.

```
$i = 1;  
until ($i > 100) {  
    print "$i ";  
    ++$i;  
}
```



do-while Statement

- The do-while statement repeats a block once before testing the condition.

```
$i = 1;  
do {  
    print "$i ";  
    ++$i;  
}while ($i <= 100);
```



for Statement

- The *for* statement repeats a block a given number of times.

```
for ($i=1; $i<=100; ++$i)
{
    print "$i ";
}
```




The localtime Function

- The **localtime** function returns the actual local time of the hosting server. It is very practical to store the result in an array. The **gmtime** function is similar and returns the Greenwich Mean Time (GMT).

```
@time = localtime;  
foreach $elem (@time)  
{  
    print "$elem ";  
}
```

sec	min	hr	d	m	y	wday	yday	dst
49	32	11	4	9	117	3	276	1

See in action here:

ihypres.net/programming/perl/exec.php?script=05
www2.scs.ryerson.ca/~dhamelin/cgi-bin/time.cgi



Array Items

- The loop on the previous page can be done by referencing individual elements:

```
@time = localtime;  
$i = 0;  
while ($i < @time) {  
    print "$time[$i] ";  
    ++$i;  
}
```

- Length of array @time
- Can also use `<= $#time`, (the last subscript)



Other Array Operations

- You can copy the first element of an array into a scalar.

```
($variable) = @array;
```

- Or the first two into 2 scalars:

```
($var1, $var2) = @array;
```

- Or the first two into scalars, the rest into another array.

```
($var1, $var2, @other) = @array;
```



Other Array Operations

- You can copy selected items into another array:

```
@choice = @array[2,5,$i];
```

- You can add an element to the end of the array:

```
push (@array, $new_element);
```

- You can add an element at the beginning of the array:

```
unshift (@array, $new_element);
```



Other Array Operations

- You can remove the last item of an array:
- You can remove the first item of an array:

```
$removed = pop (@array);
```

```
$removed = shift (@array);
```

- You can sort an array by ASCII order:

```
@array = sort (@array);
```

- You can reverse an array:

```
@array = reverse (@array);
```

|



Array Sorting

- The simple sort function sorts by ASCII values. There are variations.
- To sort by reverse ASCII order:

```
@array = sort{$b cmp $a} (@array);
```

- To sort numerically in ascending order:

```
@array = sort{$a <=> $b} (@array);
```

- To sort numerically in descending order:

```
@array = sort{$b <=> $a} (@array)
```



Subroutines

- A subroutine is a block of statements that will be executed when called.
- **sub** creates a new subroutine. Here is a simple one.

```
sub mime {  
    print "Content-type:  
    text/html\n\n";  
}
```

- It is simply called like this: **mime () ;**



Subroutine Arguments

- Subroutines can have arguments. You don't have to specify them though.
- The values passed in all come packaged up in the array `@_`.

```
sub f1 {  
    print "$_[0]";  
}
```

```
$s = "Hello World!";  
f1($s);
```



Subroutine Return Value

- A Perl subroutine can return a single final value or return value.
- The return value can be a scalar, an array or a hash.
- The return value is either the value of the last expression evaluated in the subroutine or any explicit value returned by the **return** statement.
- A subroutine is very similar to a C function.



Subroutine Return Value

```
sub f1 {  
    $x = 10 - $_[0];  
    $y = 5 - $_[0];  
    return ($x);  
}
```

```
print f1(3);
```

- *With return (\$x), the answer is 7, without it the answer is 2.*



External Subroutines

- It is very common to store subroutines in external files. Each subroutine file must end with **1**;
- The main program will use the `require` statement to link to the subroutine file(s).

```
require 'external.lib' ;
```



Hashes

- A hash is a structure that contains pairs of associated elements. They are also called associative arrays in other programming languages.
- Assign key/values to a hash:

```
%tiger_data = ('English name' => 'Tiger', 'Latin name' => 'panthera tigris', population => '4500', status => 'endangered');
```



Hashes: Accessing Values

- To access a single value from a hash, we use `$hash{key}`:

```
print "The status of the  
$tiger_data{'English name'}  
is $tiger_data{'status'}.";
```

The status of the Tiger is endangered.



Operations on Hashes

- Replacing a value:

```
$tiger_data{'population'} =  
4000;
```

- Getting several values (taking a slice off the hash):

```
@tiger_data{'English name', 'Latin  
name'}
```

OR

```
@arr = ('English name', 'Latin  
name');  
@tiger_data{@arr};
```




Operations on Hashes

- Getting all the keys:

```
@keys = keys (%tiger_data);
```

- Getting all the values:

```
@values = values (%tiger_data);
```

- Getting all the pairs:

```
while (($key, $value) =  
    each (%tiger_data)) {  
    print "<p>Their $key is  
$value.</p>";  
}
```



Operations on Hashes

- Ordering the pairs:

```
foreach $key (sort (keys  
    %tiger_data)) {  
    print "<p>Their $key is  
    $tiger_data{$key}.</p>";  
}
```

- Removing a value:

```
$removed_value = delete  
    $tiger_data{'population'};
```

- Checking if a key exists:

```
if (exists $tiger_data{'color'})  
    . . .
```



Analysing Data

- One of the most powerful features of the Perl programming language is its strength for decomposing and analysing data strings. The three main operations are called *match*, *substitute* and *split*.
- **Match**: checks if a variable contains a specific data.
- **Substitute**: checks and changes the specific data for something else.
- **Split**: divides a variables into multiple parts.



Match

- `=~ m/stringtofind/i` is the match operator. By adding `i` after the last forward slash, you indicate that the search is not case sensitive.

```
$s = 'Life is good!';  
if ($s =~ m/good/) {  
    print "The word good is  
there";}  
else{  
    print "The word good is  
not there";}
```



Substitute

- `=~s/oldstring/newstring/ig` is the substitute operator. By adding `i` after the last forward slash, you indicate that the search is not case sensitive. By adding `g`, the substitution is global (all occurrences) not just the first one.

```
$s = 'Life is good!';  
$s =~ s/good/great/gi;  
print $s;
```

- *Note: After a match, `$&` contains the replaced value (good) and `$'` the string that follows the matched part (!).*



Split

- The split function splits a scalar variable into pieces. It is particularly useful for data records.

```
$rec = '3434;Hammer;25;29.95';  
@pieces = split (//;/,$rec);
```

```
foreach $piece (@pieces) {  
    print "<p>$piece</p>";  
}
```

See in action here:

ihypress.net/programming/perl/exec.php?script=13



Regular Expressions

- Regular expressions are used to configure search patterns. See the course website and the following pages for details on regular expressions in Perl.
- www.cs.tut.fi/~jkorpela/perl/regexp.html
- www.enginsite.com/Library-Perl-Regular-Expressions-Tutorial.htm



Some Examples of Regular Expressions

```
$mystring = "The phone number is 555-2345";  
if($mystring =~ m/(\d)/) {  
    print "The first digit is $1.";}  
The first digit is 5.
```

In order to designate a pattern for extraction, one places parenthesis around the pattern. If the pattern is matched, it is returned in the Perl special variable called \$1. If there are multiple expressions in parentheses, then they will be in the variables \$1, \$2, \$3, etc...



Some Examples of Regular Expressions

```
$mystring = "The phone number is 555-  
2345" ;
```

```
if($mystring =~ m/(\d+)/) {  
    print "The first number is  
    $1." ;}
```

The first number is 555.

A number here really means a grouping of one or more digits. The pattern quantifier + matches one or more of the pattern that immediately precedes it, in this case, the \d. The search will finish as soon as it reads the "555".



Some Examples of Regular Expressions

```
$mystring = "The phone number is 555-  
2345";
```

```
while($mystring =~ m/(\d+)/g) {  
    print "Found number $1. " ;}
```

Found number 555. Found number 2345.

The pattern modifier **g** tells Perl to do a global search on the string. In other words, search the whole string from left to right.



Some Examples of Regular Expressions

```
$mystring = "The start text always precedes  
the end of the end text.";
```

```
if($mystring =~ m/start(.*?)end/) {  
    print $1;}  
}
```

text always precedes the end of the

The pattern `.*` is two different metacharacters that tell Perl to match everything between the words start and end. Specifically, the metacharacter `.` means match any symbol except new line. The pattern quantifier `*` means match zero or more of the preceding symbol.



Some Examples of Regular Expressions

```
$mystring = "The start text always precedes  
the end of the end text.";
```

```
if($mystring =~ m/start(.*?)end/) {  
    print $1; }
```

text always precedes the

By default, the quantifiers are greedy. This means that when you say `.*`, Perl matches every character (except new line) all the way to the end of the string, and then works backward until it finds **end**. To make the pattern quantifier miserly, you use the pattern quantifier limiter `?`. This tells Perl to match as few as possible of the preceding symbol before continuing to the next part of the pattern.



Remember Submitted Information

- Hidden fields are one way to remember information submitted by visitors. Cookies are another one.
- Hidden fields are used to keep information from an earlier form. Suppose you have two forms to gather data. How can we keep the information from the first form when submitting the second form?
- The answer: fields of the first form's data are hidden in the second form.



Hidden Fields

- Note however that hidden fields, although hidden on the browser window are still visible in the HTML source code!

```
<input type="hidden" name="name"  
value="value">
```

- The CGI.pm library contains lots of extra options to work with hidden fields and more, the complete documentation is at this address:

www.wiley.com/legacy/compbooks/stein/



Hidden Fields

- Here is an example of a use of hidden fields.

1. The HTML form

```
<form  
  action="http://www2.scs.ryerson.  
  ca/~dhamelin/cgi-bin/form1.cgi"  
  method="post">
```

What is your first name?

```
<input type="text" name="name" />  
<input type="submit" />  
</form>
```



Hidden Fields

2. The first cgi program

```
#!/usr/bin/perl
use CGI ':standard';
print "Content-type: text/html\n\n";

$fn = param ('name');
print "<p>Hello $fn!</p>";

print qq(<form action="form2.cgi"
        method="post">);
print qq(<input type="hidden" name="first"
        value="$fn">);
print "Where are you from?";
print qq(<input type="text" name="city">);
print qq(<input type="submit"></form>);
```



Hidden Fields

3. The second cgi program

```
#!/usr/bin/perl
use CGI ':standard';
print "Content-type: text/html\n\n";

$fn = param ('first');
$city = param ('city');

print qq(<p style="text-align:center;font-
size:36px;color:blue">Hello $fn from
$city!</p>);
```

See in action here:

www.scs.ryerson.ca/dhamelin/course_stuff/cps530/form0.html



Cookies

- Text files downloaded onto a visitor's computer hard drive to store the visitor's actions in order to better customise their following visits.
- Hold information on the times and dates you have visited web sites. Other information can also be saved to your hard disk in these text files, including information about online purchases, validation information about you for members-only web sites, and more.



Cookies

- Sending a cookie involves collecting the information to save, giving it a name and send it to the visitor's browser.
- 1. Gathering information (HTML form):

```
<form action="http://www2.scs.ryerson.ca/~dhamelin/cgi-  
bin/cookie1.cgi" method="post">  
Select a greeting language:<br>  
<input type="radio" name="lang" value="en"> English<br>  
<input type="radio" name="lang" value="fr">  
Fran&ccedil;ais<br>  
<input type="radio" name="lang" value="es">  
Espa&ntilde;ol<br>  
<input type="radio" name="lang" value="it"> Italiano<br>  
<input type="radio" name="lang" value="tu"> Turkce<br>  
<input type="submit">  
</form>
```



Cookies

- 2. Sending the cookie:

```
#!/usr/bin/perl
use CGI ':standard';
$lang = param ('lang');
print "Set-Cookie:language=$lang\n";

print "Content-type: text/html\n\n";
print "<p>Hello</p>";
print "<p>On your next visit. I will
      greet you in the language of your
      choice.</p>";
```

See in action here:

www.scs.ryerson.ca/dhamelin/courses/cps530/cookie0.html



Cookies

- 3. Reading the cookie:

```
#!/usr/bin/perl
use CGI ':standard';
print "Content-type: text/html\n\n";

%greeting = (en=>'Hello',fr=>'Allo',es=>'Hola',it=>'Buon
giorno',tu=>'Merhaba');
$language = cookie('language');

if ($language){
    print qq(<p style="font-
size:60px">$greeting{$language}</p>);
}
else {
    print qq(<p style="font-size:60px">Hello<br>(no
language preference found)</p>);
}
```

See in action here:

www2.scs.ryerson.ca/~dhamelin/cgi-bin/cookie2.cgi



Cookie Options

- Setting an expiration date:

```
print "Set-  
Cookie:language=$lang;expires=3-  
May-2027 00:00:00 GMT\n";
```

- Limiting a cookie to a domain:

```
print "Set-  
Cookie:language=$lang;domain=scs.ry  
erson.ca\n";
```

- Limiting a cookie to a directory:

```
print "Set-  
Cookie:language=$lang;domain=scs.ry  
erson.ca;path=/~dhamelin\n";
```



String Case Functions

- Some functions make it easy to change the case characters in a string.
- **uc**: all the characters in upper-case
- **ucfirst**: the first character only in upper-case.
- **lc**: all the characters in lower-case
- **lcfirst**: the first character only in lower-case.

Ex: `$newstring = uc ($oldstring);`



HTML Shortcut

- Many times, you need to print multiple lines of HTML. Using a different print for each is time consuming. There is a better way:

```
print <<"HTML CODE";
```

```
<p>This is the<br>  
HTML code</p>
```

```
HTML CODE
```



Security Issues

- Here a few rules to maintain website security when doing server-side programming:
 1. Never trust the user to enter the proper input.
 2. Avoid system calls, especially from user input.
 3. Limit access to files (hide your directories content).
 4. Hide passwords!
 5. Be careful of programs that consume too much CPU (you may be kicked out of your host in shared hosting).



Working with Files

- In order to access a file in Perl, we need to use the open function. For example, let's suppose we want to create a new file named *data.txt*. We use the following line of code to create the file:

```
open DATA, ">data.txt";
```

- Then, to write data to the file, we use the print command:

```
print DATA "H33, Hammer, 19.99\n";  
close DATA; #we close the file.
```



Working with Files

- If the file already exists and we wish to add data to it, we open it in append mode:

```
open DATA, ">>data.txt";
```

- Then we write to it again using the print command:

```
print DATA "S43,Screwdriver,9.79\n";
```

```
close DATA; #we close the file.
```

- That's all!



Working with Files

- To read (extract data) from an existing file, we open it in read mode:

```
open DATA, "data.txt";
```

- Once we've opened the file, we can read stuff from it using the line input operator `<>`. It returns the next line from the file.

```
$line1 = <DATA>;
```

```
$line2 = <DATA>;
```

- Or you can read the whole file into an array (one line per cell):

```
@whole_file = <DATA>;
```




Working with Files

- Perl can also be used to manipulate files and work with directories.
- To delete a file:

```
unlink "data.txt";
```

- It's often useful to be able to read the contents of a directory (folder) on the server, to see what files or folders are inside it:

```
opendir FOLDER, ".";
```

```
@contents = readdir FOLDER;
```

```
closedir FOLDER;
```



Working with Folders

- Lists all the items in a folder:

```
foreach $filename (@contents)
{
    if (-d $filename) {print
"$filename [FOLDER]<br>";} else
{print "$filename [FILE]<br>";}
}
```



Uploading Files

- Would you like to give your visitors the ability to upload files to your site? It's very common and very easy to do.
- File upload works by using a special type of form field called **file**, and a special type of form encoding called **multipart/form-data**.


```
<form action="/cgi-bin/upload.cgi"
method="post" enctype="multipart/form-
data">
<input type="file" name="picture" />
<input type="submit" name="Submit"
value="Send" />
</form>
```



Uploading Files

- Next we need to write the upload.cgi script.
- Handling the data that the browser sends when it uploads a file is quite a complex process. Fortunately, the Perl CGI library, CGI.pm, does most of the dirty work for us.
- See this tutorial for using upload files in Perl:

[www.ihypress.net/programming/perl/
upload.html](http://www.ihypress.net/programming/perl/upload.html)



End of lesson