



# Lesson #6: CGI/Perl III



Subroutines

Hashes

Data Analysis

Hidden Fields & Cookies

Formatting

Security

Files & Directories



# 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.
- 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'};
```



# Operations on Hashes

- 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;
```

- 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.

```
$rec = '3434;Hammer;25;19.99';  
@pieces = split (//,$rec);  
  
foreach $piece (@pieces) {  
    print "<p>$piece</p>";  
}
```



# Regular Expressions

- Regular expressions are used to configure search patterns. See the course website and this following page for details on regular expressions in Perl.
- [www.cs.tut.fi/~jkorpela/perl/regexp.html](http://www.cs.tut.fi/~jkorpela/perl/regexp.html)



# Some Examples of Regular Expressions

```
$mystring = "The event took place in  
2010";
```

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

Prints *The first digit is 2*. 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 event took place in  
2010";
```

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

Prints *The first number is 2004*. 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 "2010".



# Some Examples of Regular Expressions

```
$mystring = "The event took place in  
October 20th, 2010";
```

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

Prints *Found number 20. Found number 2010.* 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;}  
}
```

Prints *text always precedes the end of the* . The pattern `.*` is two different metacharacters that tell Perl to match everything between the 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;  
  
}
```

Prints *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.



# Hidden Fields

- 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/](http://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 it in action here:

[www2.scs.ryerson.ca/~dhamelin/cps530/form0.html](http://www2.scs.ryerson.ca/~dhamelin/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:

```
<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" />
    Français<br>
  <input type="radio" name="lang" value="es" /> Español<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>";
```



# 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>);
}
```



# Cookie Options

- Setting an expiration date:

```
print "Set-  
Cookie:language=$lang;expires=3-  
May-2007 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";
```



# Formatting Output

- Like in C, it is possible to format the output in Perl. Here are a few examples:

```
printf ( "%.2f", $price );
```

```
$formatted_price = sprintf ( "%8.2f",  
    $price );
```

```
printf ( "%-5d", $n );
```

```
printf ( "%08d", $n );
```

```
printf ( "%-20s", $string );
```

- Remember that extra spaces are ignored in HTML!



# 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.

```
$newstring = uc ($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:
  - 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).



# Working with Files

Visit this website:

[www.elated.com/tutorials/programming/perl\\_cgi/working\\_with\\_files/](http://www.elated.com/tutorials/programming/perl_cgi/working_with_files/)



# Uploading Files

Visit this website:

[www.sitepoint.com/article/uploading-files-cgi-perl](http://www.sitepoint.com/article/uploading-files-cgi-perl)



# End of lesson