Applied Perl

Boston University
Office of Information Technology

Course Number: 4095

Course Coordinator: Timothy Kohl

Last Modified: 08/31/07

Outline

- Perl as a command line tool
- Perl in system administration
- Perl and SQL (MySQL)
- GUI's in Perl (Perl-Tk)
- Perl and the Web

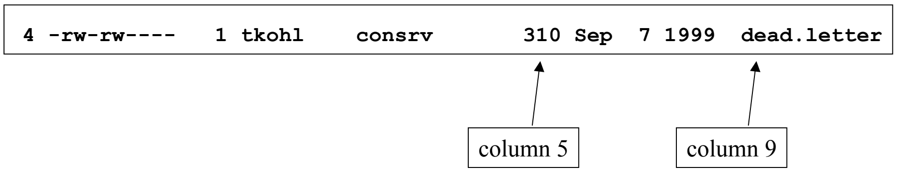
Perl as a command line tool.

Although the primary mechanism for using Perl is through scripts, Perl can be used on the command line in conjunction with other programs using Unix pipes.

Ex: Take the output of 'ls -als' and print the file names and sizes only.
Typically, the output of ls -als looks like this.

```
4 -rw-rw----   1 tkohl    consrv       310 Sep  7 1999  dead.letter
```

The point being, that (if we number the columns from left to right, starting with **0)** then the two columns of interest are as shown.



```
4 -rw-rw----   1 tkohl    consrv       310 Sep  7 1999  dead.letter
```

The command sequence would be as follows:

```
>ls -als | perl -ane 'print "$F[5] $F[9]\n"'
```

How does this work?

```
>ls -als | perl -ane 'print "$F[5] $F[9]\n"'
```

-e   execute the code in quotes
-n   execute the code for every line of standard input
     (i.e. assume a while(<STDIN>) loop has been wrapped around
          the code to execute, with each line assigned to **`$_`**)
-a take the line of standard input and let
    `@F=split(/\s+/,$_)`

The effect is that the output of

ls -als

is split into columns, and then we print out the columns of interest (5 and 9)

Perl's regular expression matching can be put to use on the command line.

Ex: Your Unix path is given by the environmental variable **`$PATH`**

```
>echo $PATH

.:/home/tkohl/bin:/usr/vendor/bin:/usr/local/4bin:
/usr/local/bin:/usr/ucb:/usr/bin:/usr/bin/X11
```

If you want a more readable list, you can do the following:

```
>echo $PATH | perl -ne 's/:/\n/g;print'
```

print the result

take the path separated by **:**

replace **every** occurrence of **:** with a newline **\n (note we are acting on the variable $_)**

The result then is

```
.
/home/tkohl/bin
/usr/vendor/bin
/usr/local/4bin
/usr/local/bin
/usr/ucb
/usr/bin
/usr/bin/X11
```

We can even shorten this by using the **`-p`** option which automatically prints the variable **`$_`**

```
>echo $PATH | perl -pne 's/:/\n/g'
```

We can also do in-place modification of a file using Perl on the command line.

Ex: Say we wish to replace every occurrence of the word 'Foo' in the file called **somefile** by the word 'Bar'

```
>perl -p -i.old -e 's/Foo/Bar/g' somefile
```

use the print option
to print the contents of $_

substitution to apply
everywhere (g option)

-e means execute this code

file to modify

-i (**in place operation**)
and do the modifications to
a file called **somefile.old**
and then copy it back to the
original **somefile**

Perl as a system administrator's tool.

In this section we examine Perl's role in system administration.

As many of the files that control the behavior of a Unix system are text files, and since Perl excels at text file processing it is a natural choice for system administrators.

There is also the fact that it takes less time to assemble a Perl script to do a certain task than, say, a corresponding C program to do the same thing.

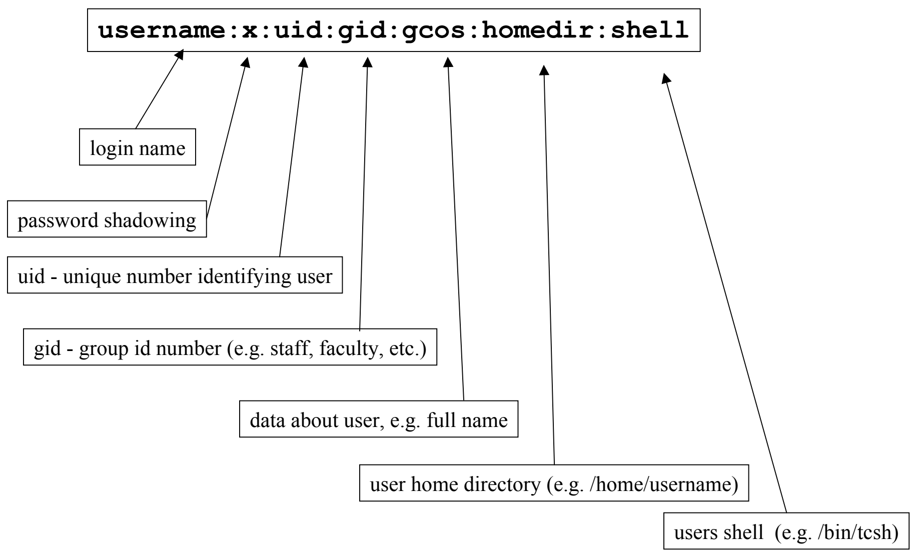
Problem: To lock the accounts of users who have not logged in within the last 6 months.

Tactic: Check the age (access time) of the .login file in each users home directory.

First, how do we get a list of all the 'ordinary' users on the system. Usually, there are two files of importance,

/etc/passwd	contains user information
and	
/etc/shadow	contains encrypted passwords (not needed)

The structure of /etc/passwd looks like this.



```
username:x:uid:gid:gcos:homedir:shell
```

login name
password shadowing
uid - unique number identifying user
gid - group id number (e.g. staff, faculty, etc.)
data about user, e.g. full name
user home directory (e.g. /home/username)
users shell (e.g. /bin/tcsh)

Ex:

```
fred:x:3216:25000:Fred Flintstone:/home/fred:/bin/bash
```

3216 is fred's uid and 25000 is his gid.

As such, there may be others with the same gid (i.e. belong to the same group) but only one with that uid.

Since we are interested in looking at the accounts of ordinary users which have only certain types of uids and gids we can, for example, restrict our attention to those in the password file with certain gids

Ex:

25000	- students
25001	- faculty
25002	- staff

For users with one of these gid's we will check to see if they logged in sometime the last 6 months and, if not, lock their account.

So we need to parse the /etc/passwd file and grab the entries with those gid's of interest.

Ex: Let's first look at the password file and print out those lines with one of the gid's we're looking for.

```
username:x:uid:gid:gcos:homedir:shell
```

```
#!/usr/local/bin/perl5
@GID=("25000","25001","25002");
open(P,"/etc/passwd");
while($line=<P>){
	chomp($line);
	@fields=split(/:/,$line);
	foreach $gid (@GID){
		if ($fields[3] ==$gid){
			print "$line\n";
		}
	}
}
close(P);
```

gid's of interest

open password file

split up each line along **:** and assign to array **@fields**

loop over **@GID** and check

close password file

Ok, now what?

Contained in each line is the home directory of the given user, say /home/username

As such, their .login file is

/home/username/.login

To check the access time, of this file, we can use the **`-A`** file test operator which returns the number of days since the given file (or directory) was accessed.

So we will use a conditional of the form:

```
if(-A "/home/username/.login" >180){
      # lock their account
}
```

So, here is how the final script might go.

```
#!/usr/local/bin/perl5
@GID=("25000","25001","25002");
$noshell="/bin/nosh";   # void shell prevents login
system("cp /etc/passwd /etc/passwd.save"); # safety first!
open(P,"/etc/passwd");
open(NP,">/etc/newpasswd");
while($line=<P>){
       chomp($line);
       @fields=split(/:/,$line);
       foreach $gid (@GID){
              if ($fields[3] ==$gid){
                     $homedir=$fields[5];
                     if(-A "$homedir/.login" > 180){
                            $line=~s/$fields[6]/$noshell/;
                     }
              }
       }
       print NP "$line\n";
}
close(P);
close(NP);
system("rm /etc/passwd;mv /etc/newpasswd /etc/passwd");
```

Let's break this down.

set up gid array

setting a user's shell to /bin/nosh makes logins impossible

```
#!/usr/local/bin/perl5
@GID=("25000","25001","25002");
$noshell="/bin/nosh";
system("cp /etc/passwd /etc/passwd.save");
open(P,"/etc/passwd");
open(NP,">/etc/newpasswd");
```

make a backup of /etc/passwd

this is the modified version of /etc/passwd

```
while($line=<P>){
        chomp($line);
        @fields=split(/:/,$line);
```

Read in /etc/passwd one line at time and split the fields up along **:**

check each line for one of the gid's we want

pick out home dir.

check if .login has not been accessed for over 180 days

```
foreach $gid (@GID){
       if ($fields[3] ==$gid){
              $homedir=$fields[5];
              if(-A "$homedir/.login" > 180){
                   $line=~s/$fields[6]/$noshell/;
              }
       }
   }
   print NP "$line\n";
```

if so, then replace shell (**`$fields[6]`**) with "/bin/nosh"

regardless of whether we modified the user's shell, write the line to the file /etc/newpassd

```
}
close(P);
close(NP);
system("rm /etc/passwd;mv /etc/newpasswd /etc/passwd");
```

Once done, close both /etc/passwd, and /etc/newpasswd

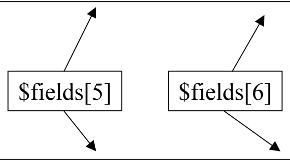
Then remove the old /etc/passwd and replace it with the modified version.

Note, we made a backup of /etc/passwd beforehand in case something went wrong while this script was running.

To clarify, if /home/fred/.login has not been accessed for more than 6 months then this what happens to his entry in /etc/passwd

```
fred:x:3216:25000:Fred Flintstone:/home/fred:/bin/bash
```

becomes



```
fred:x:3216:25000:Fred Flintstone:/home/fred:/bin/nosh
```

Perl and SQL (MySQL)

A SQL database is a database that can be interacted with via a command syntax known as Structured Query Language.

Essentially, one asks questions of the database, questions phrased in such a way as to return very precise information from the tables in the database.

The database itself consists of fields with certain labels where each entry has specified values for these fields.

Ex: Suppose you have a database of books called Books where the fields are say

TITLE
AUTHOR
DATE
PAGES

then you could do the following

```
>use Books;
>select TITLE from Booklist;

Hamlet
The Gold Bug
The Stranger
.
.


300 rows in set (0.01 sec)

>
```

Here we're assuming that the database is called 'Books' and the table containing the fields is called 'Booklist'

So here we are asking for the TITLE fields for all the entries in the table Booklist.

diagnostic message from the database server

One could also query multiple fields

```
>use Books;
>select TITLE,AUTHOR from Booklist;

Hamlet                  William Shakespeare
The Gold Bug            Edgar Allan Poe
The Stranger            Albert Camus
.
.
.

300 rows in set (0.01 sec)

>
```

The real power of SQL is in the ability to make more complicated queries.

```
>use Books;
>select TITLE,AUTHOR from Booklist where AUTHOR like 'Albert Camus';

The Plague              Albert Camus
The Stranger            Albert Camus
.
.
.

19 rows in set (0.01 sec)

>
```

i.e. Return the TITLE and AUTHOR fields where AUTHOR=Albert Camus

There are quite a number of implementations of SQL databases, the one I will be using for my examples is MySQL.

Regardless of which implementation, the real point in all this is that there is a Perl module called DBI which one can use to transact with SQL databases of varying types from within a Perl script without the need to resort to some **`system()`** or **`` `command` ``** mechanism.

I will give a tiny example, re-using the Book database.

```
#!/usr/local/bin/perl5
use DBI;
$db=DBI->connect('DBI:mysql:Books:localhost','me',undef);
$query="select TITLE,AUTHOR from Booklist";
$table_data=$db->prepare($query);
$table_data->execute;
$table_data->bind_columns(undef,\($Title,$Author));
while($table_data->fetch){
    print "[$Title $Author]\n";
}
```

`use DBI;` We will be using the DBI.pm module

```
$db=DBI->connect('DBI:mysql:Books:localhost','me',undef);
```

We create a DBI object ($db) by connecting (as user 'me') to an existing MySQL server running on localhost (i.e. the machine we're logged into) and indicate that we wish to access the Books database

```
$query="select TITLE,AUTHOR from Booklist";
$table_data=$db->prepare($query);
$table_data->execute;
```

`$query` is the query we wish to pass to the database we're now connected to, and **`$table_data`** will be where we get the result returned by the server *once the query is executed*

don't worry about this

```
$table_data->bind_columns(undef,\($Title,$Author));
while($table_data->fetch){
    print "[$Title $Author]\n";
}
```

Here we bind the columns that will be returned from the query to two variables **`$Title`** and **`$Author`** while there are results to be 'fetched' from the database and printed.

Can you guess what kind of object **`\($Title,$Author)`** is ?

Ex:
```
[Hamlet                  William Shakespeare]
[The Gold Bug            Edgar Allan Poe]
[The Stranger            Albert Camus]
.
.
```

and so on.

Doing structured queries is just as easy.

```
#!/usr/local/bin/perl5
use DBI;
$db=DBI->connect('DBI:mysql:Books:localhost','me',undef);
$query="select TITLE,AUTHOR from Booklist where
        TITLE like \'The Stranger\'";
$table_data=$db->prepare($query);
$table_data->execute;
$table_data->bind_columns(undef,\($Title,$Author));
while($table_data->fetch){
    print "[$Title $Author]\n";
}
```

The only difference is the line

```
$query="select TITLE,AUTHOR from Booklist where TITLE like \'The Stranger\'";
```

which is exactly like the command line SQL query seen earlier except we need to escape the `'` with `\'` when creating **`$query`**

This is just the tip of the iceberg. One can not only make queries but also modify or create a database using DBI.

To read more about DBI, consult the references at the end of the tutorial.

GUI's in Perl (Perl-Tk)

There exists another scripting language (actually two) which came out around the same time as Perl, known as Tcl (tool command language) and Tk (toolkit).

Tcl is still around today but it is the counterpart language Tk that attracted the interest of many GUI (graphical user interface) developers.

The reason being that it allowed one to create all manner of GUI 'widgets' like windows, sliders, buttons but in a scripting language.

In the Unix world, before this, one generally had to use specialized (and very cryptic) C libraries to create applications with GUIs.

Something as simple, for example, as creating a window with a button that did something when clicked, was a highly non-trivial task.

So what does this have to do with Perl?

Well, people liked Tk so much that they incorporated its (object oriented) functionality into Perl by creating a module called Tk.pm

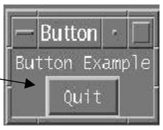
Now, the original syntax of Tk had a very hierarchical feel, in that it allowed one to create window 'objects' in a top down fashion. Moreover, it allowed one to have 'events' such as mouse clicks trigger functions in one's program.

These same features have carried over into the Perl implementation but the syntax is now thoroughly Perl.

Ex: (extremely simple!)

```
#!/usr/local/bin/perl5
use Tk;
my $main = MainWindow->new();
$main->Label(text=>"Button Example")->pack;
$main->Button(text=>"Quit",command=>sub{exit})->pack;
MainLoop;
```

Clicking the button here will make the program terminate and the window go away.



```
use Tk;
```

incorporate in the Tk.pm module

```
my $main = MainWindow->new();
```

create a window object using the **new()** constructor in Tk.pm

```
$main->Label(text=>"Button Example")->pack;
$main->Button(text=>"Quit",command=>sub{exit})->pack;
```

Here we create two objects a Button and a Label which are contained within **$main**, the window object we started with.

The Label is, in fact, another object which includes an associative array of attributes, for instance the text indicated by

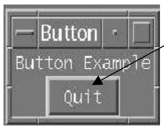
```
text=>"Button Example"
```

The Button object is also contained within **`$main`** and also has an associative array of attributes.

Moreover, one of its attributes **`command`**, is associated to a function call (in this case an anonymous subroutine) that exits the program when the button is clicked.

```
$main->Button(text=>"Quit",command=>sub{exit})->pack;
```

anonymous sub corresponds to clicking the button



The **`pack()`** command is what brings draws the object on the screen.

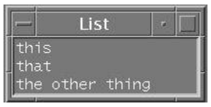
Actually what this does is to 'pack' the different components into the window that was created with **`new()`**

The command **`MainLoop`** is what keeps the program running until someone closes the open window, either by clicking the ‘Quit’ button or closing the window manually.

This kind of functionality is called ‘**Event Driven Programming** ’ since the program does not flow from top to bottom like a typical script but sits and awaits input for an indeterminate amount of time.

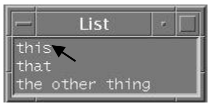
As another example, we have a 'Listbox' object.

```
#!/usr/local/bin/perl5 -w
use Tk;
$main = MainWindow->new();
$list=$main->Listbox("width"=>20,"height"=>3)->pack();
$list->insert('end',"this","that","the other thing");
$list->bind('<Double-1>',\&print_choice);

sub print_choice{
      my $choice=$list->get('active');
      return if (!$choice);
      print "$choice\n";
      $list->delete('active');
}

MainLoop();
```

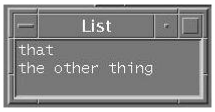


```
this
that
the other thing
```

If we now double-click on the entry '**this**' then the word 'this' gets printed on screen and the entry is removed from the list resulting in the following.



```
this
that
the other thing
```



```
that
the other thing
```

```
this
```

```
$main = MainWindow->new();
$list=$main->Listbox("width"=>20,"height"=>3)->pack();
```

Here we create a top level window object with **`new()`** and within that we create a Listbox object with size parameters specified.

```
$list->insert('end',"this","that","the other thing");
```

The **`insert()`** method acts on the **`$list`** object by inserting elements one after each other in the listbox.

```
$list->bind('<Double-1>',\&print_choice);
```

This 'binds' the action of double-clicking mouse button **1** to the function **`print_choice()`**

```
sub print_choice{
      my $choice=$list->get('active');
      return if (!$choice);
      print "$choice\n";
      $list->delete('active');
}
```

The **`print_choice()`** function takes the active entry (that which was double-clicked on) and returns the string at that position and into the variable **`$choice`** which then gets printed and is subsequently deleted from the listbox object **`$list`**

To read more about this, consult the references at the end of the tutorial.

Perl and the Web

Perl is used in many ways for web applications, including the management of web servers as well as CGI scripting and more.

Our first example will involve the analysis of web server logs.

In particular we will show how to parse the log files and retrieve the important statistical information contained therein, such as the addresses of those sites connecting to the server as well as content downloaded etc.

The basic information that is recorded in any web 'event' which a server might record are:

- the address of the incoming connection (i.e. who visited)
- the time of the connection
- what content they downloaded

Additionally, one may record other data such as:

- any site they came to yours by via a link
- the hardware/software combination they use
(e.g. Unix, Windows, Netscape, IE)

Ex: A typical entry in an access_log file:

```
168.122.230.172 - - [16/Feb/2001:08:42:52 -0500] "GET /people/tkohl/teaching/sprin
g2001/secant.pdf HTTP/1.1" 200 0 "http://math.bu.edu/people/tkohl/teaching/spri
ng2001/MA121.html" "Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)"
```

`168.122.230.172`	IP address of visitor
`[16/Feb/2001:08:42:52 -0500]`	time
`"GET /people/tkohl/teaching/spring2001/secant.pdf HTTP/1.1"`	content they retrieved
`200 0`	server response code
`"http://math.bu.edu/people/tkohl/teaching/spring2001/MA121.html"`	referrer
`"Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)"`	client software and archictecutre

```
168.122.230.172 - - [16/Feb/2001:08:42:52 -0500] "GET /people/tkohl/teaching/sprin
g2001/secant.pdf HTTP/1.1" 200 0 "http://math.bu.edu/people/tkohl/teaching/spri
ng2001/MA121.html" "Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)"
```

In order to parse this file and extract the relevant information, say for some statistical analysis or whatever, we need to describe log entries with a regular expression and extract the different components.

Here is a subroutine for parsing entries such as the one above.

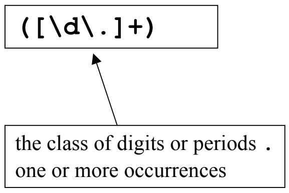
```
sub parse_log{
        my $entry = $_[0];
        $entry =~ /([\d\.]+) \- \- (\[[^\]]+\]) \"([^\"]+)\" (\d+ \d+)
\"([^\"]+)\" \"([^\"]+)\"/;
        return ($1,$2,$3,$4,$5,$6);
}
```

Let's examine the pattern to clarify what's going on.

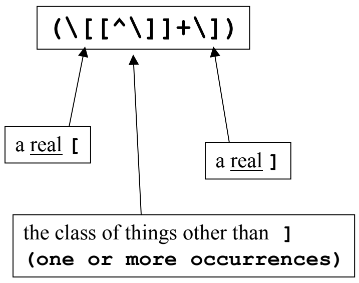
```
168.122.230.172 - - [16/Feb/2001:08:42:52 -0500] "GET /people/tkohl/teaching/sprin
g2001/secant.pdf HTTP/1.1" 200 0 "http://math.bu.edu/people/tkohl/teaching/spri
ng2001/MA121.html" "Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)"
```

Discounting the spaces and dashes between the entries, here are the patterns describing the portions to memorize.

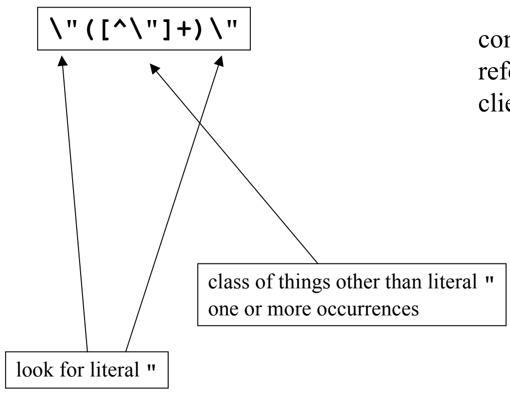
`([\d\.]+)`	**ip address**
`(\[[^\]]+\])`	**date (including the brackets**
`\"([^\"]+)\"`	**content downloaded**
`(\d+ \d+)`	**status code**
`\"([^\"]+)\"`	**referrer**
`\"([^\"]+)\"/`	**client info**



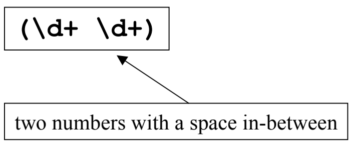
IP address



date



content downloaded
referrer
client information



status code

So now, the components of the log entry are returned as an array from the parse_log function.

So we might use it in a larger script as follows:

```
open(LOG,"/usr/local/apache/logs/access_log");
while($line=<LOG>){
        ($ip,$date,$content,$status,$referrer,$client)=parse_log($line);
        # do something with the components
}
close(LOG);
```

CGI

CGI stands for 'Common Gateway Interface' and is a method (really a collection of methods) for passing information from a client to a web server.

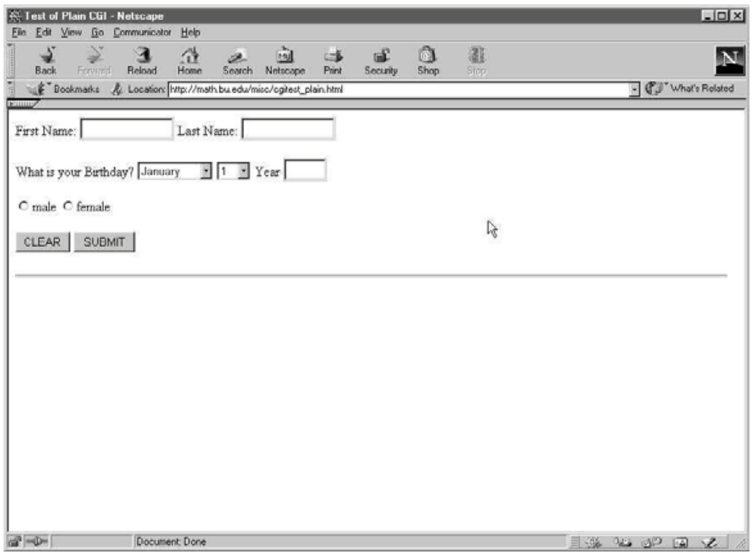
This is the primary mechanism for, amongst other things, processing fill out forms on web pages.

For example, entering a query into a search engine of some sort.

We will not be discussing the CGI mechanism in detail, but rather, illustrate a simple forms interface to a cgi script written in Perl.

Our demo will consist of a simple form for a user to enter in biographical information. This will then be submitted and displayed by the server.

The page itself will look like this.


Test of Plain CGI - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://math.bu.edu/misc/cgitest_plain.html
What's Related
Document: Done

First Name: Last Name:

What is your Birthday? January 1 Year

male female

CLEAR SUBMIT

There will be 6 variables passed from the form to the CGI script for processing.

`firstname`
`lastname`
`month`
`day`
`year`
`gender`

Here is the source.

```
<HTML>
<TITLE>
Test of Plain CGI
</TITLE>
<BODY>
<FORM METHOD="GET" ACTION="/cgi-bin/cgitest_plain">
First Name:
<INPUT TYPE="text" NAME="firstname" VALUE="" MAXLENGTH=20 SIZE=10>
Last Name:
<INPUT TYPE="text" NAME="lastname" VALUE="" MAXLENGTH=20 SIZE=10>
<P>
What is your Birthday?

<SELECT NAME="month">
<OPTION VALUE="January" SELECTED> January
<OPTION VALUE="February"> February
<OPTION VALUE="March"> March
<OPTION VALUE="April"> April
<OPTION VALUE="May"> May
<OPTION VALUE="June"> June
<OPTION VALUE="July"> July
<OPTION VALUE="August"> August
<OPTION VALUE="September">  September
<OPTION VALUE="October"> October
<OPTION VALUE="November"> November
<OPTION VALUE="December"> December

</SELECT>
```

script which processes data from form

firstname and lastname

month they were born

```
<SELECT NAME="day">
<OPTION VALUE="1" SELECTED> 1
<OPTION VALUE="2"> 2
<OPTION VALUE="3"> 3
<OPTION VALUE="4"> 4
<OPTION VALUE="5"> 5
<OPTION VALUE="6"> 6
<OPTION VALUE="7"> 7
<OPTION VALUE="8"> 8
<OPTION VALUE="9"> 9
<OPTION VALUE="10"> 10
<OPTION VALUE="11"> 11
<OPTION VALUE="12"> 12
<OPTION VALUE="13"> 13
<OPTION VALUE="14"> 14
<OPTION VALUE="15"> 15
<OPTION VALUE="16"> 16
<OPTION VALUE="17"> 17
<OPTION VALUE="18"> 18
<OPTION VALUE="19"> 19
<OPTION VALUE="20"> 20
<OPTION VALUE="21"> 21
<OPTION VALUE="22"> 22
<OPTION VALUE="23"> 23
<OPTION VALUE="24"> 24
<OPTION VALUE="25"> 25
<OPTION VALUE="26"> 26
<OPTION VALUE="27"> 27
<OPTION VALUE="28"> 28
<OPTION VALUE="29"> 29
<OPTION VALUE="30"> 30
<OPTION VALUE="31"> 31
</SELECT>
```

the day of the month they were born

the year they were born

```
Year <INPUT TYPE="text" NAME="year" VALUE="" MAXLENGTH=4 SIZE=4>
<P>
<INPUT TYPE="radio" NAME="gender" VALUE="male">male
<INPUT TYPE="radio" NAME="gender" VALUE="female">female
<P>
<INPUT TYPE="reset" VALUE="CLEAR">
<INPUT TYPE="submit" VALUE="SUBMIT">
</FORM>
<HR>
```

their gender
(note it's of type **radio** which allows only one value or the other)

reset all the fields in the form

submit data and initiate script

Now, without delving too deeply into how CGI works, we note the line which references the actual script which will do the work

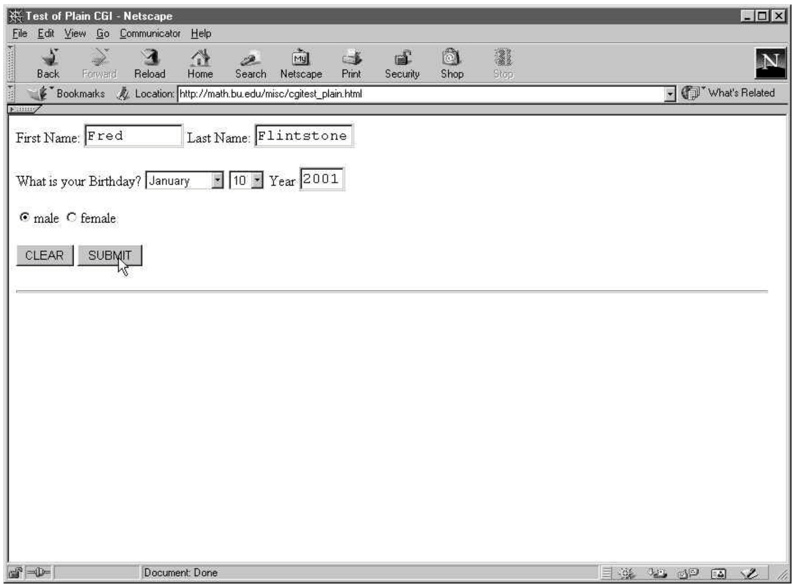
```
<FORM METHOD="GET" ACTION="/cgi-bin/cgitest_plain">
```

in particular the method **`GET`** will append the form variables to the URL of the submitted request in the following format:

`/cgi-bin/cgitest_plain?var1=value1&var2=value2&var3=value3` etc.

That is, the script will receive the form data, separated by **&** which means some processing will be necessary to extract the information.

Ex:


Test of Plain CGI - Netscape

Location: http://math.bu.edu/misc/cgitest_plain.html

First Name: Fred Last Name: Flintstone

What is your Birthday? January 10 Year 2001

[✓ male] female

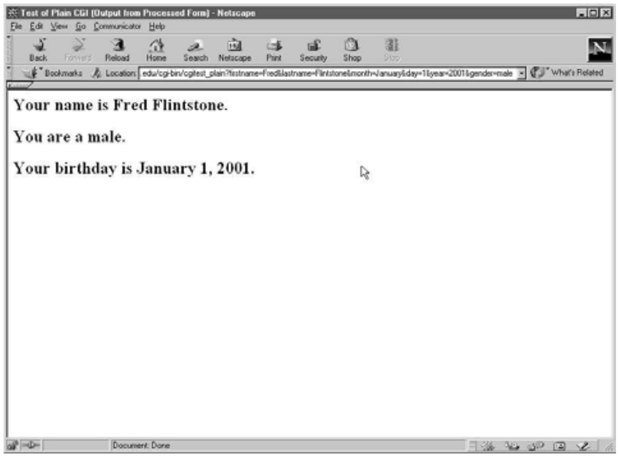
CLEAR SUBMIT

Document: Done

will result in the following URL

```
http://math.bu.edu/cgi-bin/cgitest_plain?firstname=Fred&
lastname=Flintstone&month=January&day=10&year=2001&gender=male
```

Here is the output of the script.


Test of Plain CGI (Output from Processed Form) - Netscape

Location: edu/cgi-bin/cgitest_plain?firstname=Fred&lastname=Flintstone&month=January&day=1&year=2001&gender=male

Your name is Fred Flintstone.

You are a male.

Your birthday is January 1, 2001.

Document: Done

So what about the script itself? Most commonly, these scripts are contained in the cgi-bin area of the web server.

As this is a basic example, the script isn't that long.

```
#!/usr/local/bin/perl5
$query=$ENV{'QUERY_STRING'};
foreach $pair (split(/\&/,$query)){
    ($varname,$value)=split(/=/,$pair);
    $DATA{$varname}=$value;
}
print "Content-type: text/html\n\n";
print "<HTML><HEAD>\n";
print "<TITLE>Test of Plain CGI (Output from Processed Form)</TITLE>\n";
print "</HEAD><BODY>\n";
print "<H2>Your name is $DATA{firstname} $DATA{lastname}</H2>\n";
print "<H2>You are a $DATA{gender}.</H2>\n";
print "<H2>Your birthday is $DATA{month} $DATA{day} $DATA{year} </H2>\n";
print "</BODY></HTML>\n";
```

```
$query=$ENV{'QUERY_STRING'};
foreach $pair (split(/\&/,$query)){
    ($varname,$value)=split(/=/,$pair);
    $DATA{$varname}=$value;
}
```

The **`%ENV`** associative array carries a lot of information about the user's working environment. In this case, the form data **`'QUERY_STRING'`** is passed to the script.

So here, we would have:

```
$query="firstname=Fred&lastname=Flintstone&month=January&
day=10&year=2001&gender=male"
```

To extract the form data from this string, we use the **`split()`** function

```
foreach $pair (split(/\&/,$query)){
    ($varname,$value)=split(/=/,$pair);
    $DATA{$varname}=$value;
}
```

In this case, splitting along **`&`** yields the following array to loop over with **`foreach`**

```
(firstname=Fred,lastname=Flintstone,month=January,day=10,year=2001,
gender=male)
```

We can now split each element **`$pair`** of this array into a key and value and insert it into an associative array called **`%DATA`**

Afterward, **`%DATA`** will look like this

```
%DATA = ( firstname=>Fred,
          lastname=>Flintstone,
          month=>January,
          day=>10,
          year=>2001,
          gender=>male);
```

Normally, we might put the keys above in `"` but as the names contain no special characters, we can do get away without using quotes.

Now, to display the resulting web page we proceed as follows:

```
print "Content-type: text/html\n\n";
print "<HTML><HEAD>\n";
print "<TITLE>Test of Plain CGI (Output from Processed Form)</TITLE>\n";
print "</HEAD><BODY>\n";
print "<H2>Your name is $DATA{firstname} $DATA{lastname}.</H2>\n";
print "<H2>You are a $DATA{gender}.</H2>\n";
print "<H2>Your birthday is $DATA{month} $DATA{day}, $DATA{year}.</H2>\n";
print "</BODY></HTML>\n";
```

First, to identify the output as an **html** page to the browser, we need this line.

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

Without this, the browser **won't** render the page, in fact an error code will be returned.

The rest of the script writes the html that is then rendered by the browser, with our form data included.

```
print "<HTML><HEAD>\n";
print "<TITLE>Test of Plain CGI (Output from Processed Form)</TITLE>\n";
print "</HEAD><BODY>\n";
print "<H2>Your name is $DATA{firstname} $DATA{lastname}.</H2>\n";
print "<H2>You are a $DATA{gender}.</H2>\n";
print "<H2>Your birthday is $DATA{month} $DATA{day}, $DATA{year}.</H2>\n";
print "</BODY></HTML>\n";
```

Simple Web Clients

Say one wishes to, without using a browser, download some data from a website.

Ex:

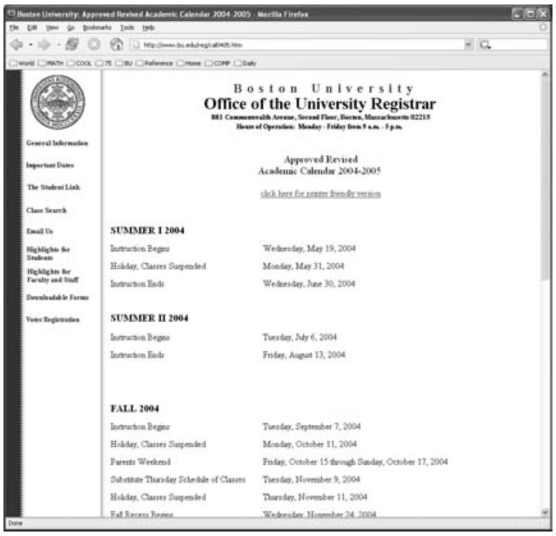
```
#!/usr/local/bin/perl5
use LWP::Simple;
print get($ARGV[0]);
```

call this ‘geturl’

```
>geturl http://www.bu.edu
```

The output will be the literal HTML code of the BU homepage, which may not be terribly interesting, but there are other ways of using such data.

Let’s consider a more interesting example.
If we wish to find the academic calendar for the 2004/5 academic year, it is located at *`http://www.bu.edu/reg/cal0405.htm`*



Boston University: Approved Revised Academic Calendar 2004-2005 - Mozilla Firefox

http://www.bu.edu/reg/cal0405.htm

Boston University
Office of the University Registrar
881 Commonwealth Avenue, Second Floor, Boston, Massachusetts 02215
Hours of Operation: Monday - Friday from 9 a.m. - 5 p.m.

General Information
Important Dates
The Student Link
Class Search
Email Us
Highlights for Students
Highlights for Faculty and Staff
Downloadable Forms
Voter Registration

Approved Revised
Academic Calendar 2004-2005

click here for printer friendly version

SUMMER I 2004

Instruction Begins	Wednesday, May 19, 2004
Holiday, Classes Suspended	Monday, May 31, 2004
Instruction Ends	Wednesday, June 30, 2004

SUMMER II 2004

Instruction Begins	Tuesday, July 6, 2004
Instruction Ends	Friday, August 13, 2004

FALL 2004

Instruction Begins	Tuesday, September 7, 2004
Holiday, Classes Suspended	Monday, October 11, 2004
Parents Weekend	Friday, October 15 through Sunday, October 17, 2004
Substitute Thursday Schedule of Classes	Tuesday, November 9, 2004
Holiday, Classes Suspended	Thursday, November 11, 2004
Fall Recess Begins	Wednesday, November 24, 2004

File Edit View Go Bookmarks Tools Help
World MATH COOL 75 BU Reference Home COMP Daily
Done

Now suppose we wish to extract the information from this page. The raw output of our script includes a lot of HTML code which certainly isn't essential information.

However, we can extract the information we want by observing that the relevant information we want lies within tags such as these

```
<TD><FONT face="Times New Roman">Instruction Begins </font></TD>
```

So we can modify our script, to, in fact, retrieve just this URL and do some custom filtering of the data.

```
#!/usr/local/bin/perl5
use LWP::Simple;
$URL="http://www.bu.edu/reg/cal0405.htm";
@DATA=split(/\n/,get($URL));
foreach (@DATA){
    if(/\<TD\>\<FONT face=\"Times New Roman\"\>(.*)\<\/font\>/){
        $item=$1;
        print "$item\n";
    }
}
```

call this getcal

which, when run yields

```
Instruction Begins
Wednesday, May 19, 2004
Holiday, Classes Suspended
Monday, May 31, 2004
Instruction Ends
Wednesday, June 30, 2004
Instruction Begins
Tuesday, July 6, 2004
Instruction Ends
Friday, August 13, 2004
.
. etc
```

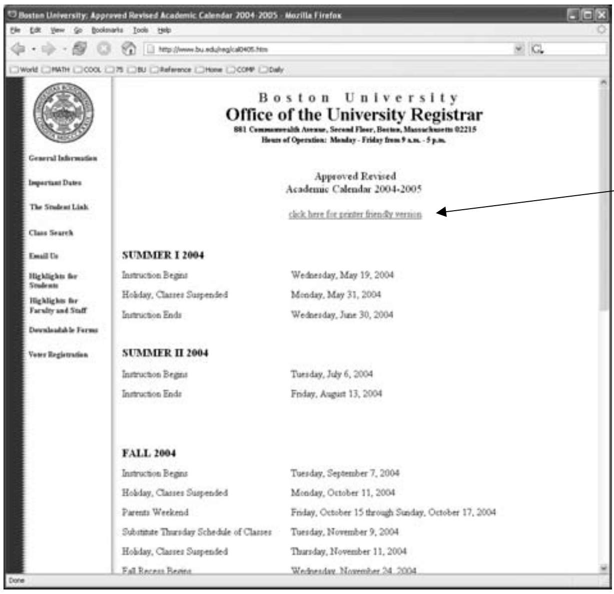
Let's add a line between each logical entry.

```
#!/usr/local/bin/perl5
use LWP::Simple;
$URL="http://www.bu.edu/reg/cal0405.htm";
@DATA=split(/\n/,get($URL));
foreach (@DATA){
    if(/\<TD\>\<FONT face=\"Times New Roman\"\>(.*)\<\/font\>/){
        $item=$1;
        print "$item\n";
        ($item=~/200(4|5)/) && (print "\n");
    }
}
```

And now the output looks a bit neater:

```
Instruction Begins
Wednesday, May 19, 2004

Holiday, Classes Suspended
Monday, May 31, 2004

Instruction Ends
Wednesday, June 30, 2004

.. Etc.
```

Of course, we could look at the original web page and observe that there is a link to a PDF version of the calendar.


Boston University: Approved Revised Academic Calendar 2004-2005 - Mozilla Firefox

http://www.bu.edu/reg/cal0405.htm

General Information
Important Dates
The Student Link
Class Search
Email Us
Highlights for Students
Highlights for Faculty and Staff
Downloadable Forms
Voter Registration

Boston University
Office of the University Registrar
881 Commonwealth Avenue, Second Floor, Boston, Massachusetts 02215
Hours of Operation: Monday - Friday from 9 a.m. - 5 p.m.

Approved Revised
Academic Calendar 2004-2005

click here for printer friendly version

SUMMER I 2004

Instruction Begins	Wednesday, May 19, 2004
Holiday, Classes Suspended	Monday, May 31, 2004
Instruction Ends	Wednesday, June 30, 2004

SUMMER II 2004

Instruction Begins	Tuesday, July 6, 2004
Instruction Ends	Friday, August 13, 2004

FALL 2004

Instruction Begins	Tuesday, September 7, 2004
Holiday, Classes Suspended	Monday, October 11, 2004
Parents Weekend	Friday, October 15 through Sunday, October 17, 2004
Substitute Thursday Schedule of Classes	Tuesday, November 9, 2004
Holiday, Classes Suspended	Thursday, November 11, 2004
Fall Recess Begins	Wednesday, November 24, 2004

Done

Perhaps we could grab just this file and put it in our home directory.

Indeed, we can!
We note that this link point to the file/URL

http://www.bu.edu/reg/images/cal0405.pdf

So….

```
geturl http://www.bu.edu/reg/images/cal0405.pdf > cal0405.pdf
```

where the '**>**' indicates we should output the result to a file in our home directory also called **cal0405.pdf**

We can then view this page at our convenience as follows:

```
acroread cal0405.pdf
```

The point in both cases is that these tools can give one the power to extract data (potentially very volatile data) from a remote site and use it in our own scripts, perhaps with a bit of filtering on our part, but this is easy when using Perl!

References for further information on Perl

Books

- Learning Perl by Randal L. Schwartz & Tom Christiansen (O'Reilly)
- Learning Perl/Tk by Nancy Walsh (O'Reilly)
- MySQL & mSQL by Yarger, Reese & King (O'Reilly)
- Programming Perl by Larry Wall, Tom Christiansen and Jon Orwant (O' Reilly)
- Programming the Perl DBI by Descartes & Bunce (O' Reilly)
- Perl in a Nutshell by Ellen Siever, Stephen Spainhour, and Nathan Patwardhan (O' Reilly)
- Official Guide to Programming with CGI.pm by Lincoln Stein (Wiley)
- Web Client Programming in Perl by Clinton Wong (O' Reilly)
- Perl for System Administration by David N. Blank-Edelman (O' Reilly)

Web

http://www.perl.com

http://www.cpan.org

http://math.bu.edu/people/tkohl/perl ← My Perl Page!

Applied Perl

Boston University
Office of Information Technology

Course Number: 4095

Course Coordinator: Timothy Kohl