Beginning CGI Programming in Perl

In this section we will lay the foundation for CGI script development.

We will introduce general CGI programming concepts relating to CGI output but then focus on Perl programming.

Specifically we will develop a very simple Perl program and see how to run it on a Macintosh and UNIX platform.

CGI Script Output

We have already mentioned that CGI scripts must adhere to standard input and output mechanism

- The **Interface** between browser and server
- Part of HTTP Protocol

For the moment we will not worry about input to a CGI script.

CGI Script Output Format

In whatever language a CGI script is programmed it **MUST** send information back in the following format:

- **The Output Header**
- **A Blank Line**
- **The Output Data**

NOTE: Between the Header and Data there **MUST** be a blank line.

CGI Output Header

- A browser can accept input in a variety of forms.
- Depending on the specified form it will call different mechanisms to display the data.
- The output header of a CGI script must specify an output type to tell the server and eventually browser how to proceed with the rest of the CGI output.

Three forms of Header Type

There are **3 forms of Header Type**:

- `Content-Type`
- `Location`
- `Status`

`Content-Type` is the most popular type.

- We now consider this further.
- We will meet the other types later.

Content-Types

The following are common formats/content-types (**there are a few others**):

Format	Content-Type
HTML	`text/html`
Text	`text/plain`
Gif	`image/gif`
JPEG	`image/jpeg`
Postscript	`application/ postscript`
MPEG	video/mpeg

Declaring `Content-Type`

To declare the `Content-Type` your CGI script must output:

`Content-Type:` **content-type specification**

Typically the `Content-Type` will be declared to produce HTML.

So the first line of our CGI script (for most of our examples) will look this:

```
Content-Type:  text/html
```

CGI Output Data

Depending on the `Content-Type` defined the data that follows the header declaration will vary:

- If it is HTML that follows then the CGI script must output **standard HTML syntax**.

Example: To produce a Web page that the server sends to a browser with a simple line of text ”Hello World!” . A **CGI script must output**:

```
Content-Type: text/html


<html>
<head>
<title>Hello, world!</title>
</head>
<body>
<h1>Hello, world!</h1>
</body>
</html>
```

Now let us see how we write and display in a Browser this CGI script in Perl

A First Perl CGI Script

Let us now look at how we write our first perl program that will be used as a CGI script.

We will learn three main things in here:

- The basic **format** of Perl CGI program
- How to **comment** Perl programs
- One Perl function `print` — which outputs data:
  - As a CGI Perl Program — Data sent to browser
  - As a stand alone Perl Program (Non- CGI) — Data sent to standard output (default: terminal window)

Format of a Perl program

Every Perl program **MUST** obey the following format:

- A first line consisting of:

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

- The rest of the program consisting of **legal** Perl syntax and commands

Strictly speaking the first line is **only required for running Perl programs on UNIX machines**.

- Since that is the intended destination of most of our Perl/CGI scripts.
- It is a good idea to **make this** the first line of every perl program.

What is the purpose of this first line?

The first line declaration has two purposes:

- It indicates that the program is a Perl script.
- It tells UNIX how to run Perl.
  Do not worry too much about this last fact — it basically specifies where in the directory hierarchy the perl interpreter program resides.
- **It MUST** be typed exactly as above to run School's UNIX/MAC OS X systems.
- The exact location **may vary** on other systems.

The first line is actually a **comment**

- Albeit a very special type of comment

Comments in Perl

It is good practice to comment all your programs (whatever the language (HTML, Perl, Java, ...) — **Suitable comments serve all programmers well**.

In Perl comments are easy:

- The # symbol indicates a comment.
- The remainder of the line is regarded as a comment
- **DO NOT** put any Perl code after a # symbol until you type a carriage return — as this will always be ignored by Perl

So a simple comment in Perl might be:

```
# hello.pl - My first Perl CGI program
```

Output from Perl

To output from a Perl script you use the `print` statement:

- The `print` statement takes a string (delimited by `"... "`) argument which it outputs.
- Similat to Java (and especially C) the string argument **formats** output.
  - You can control how the output looks from a single `print` statement.
  - The `\n` character indicates that a newline is required at that point in the text.
  - We will introduce further aspects of the `print` statement later.

First Line Output of a CGI script in Perl

For Example, The first line of our CGI script must be

- "`Content-Type: text/html`" and
- The `print` statement must have 2 `\n` characters:
  - One to terminate the current line, and
  - The second to produce the require blank line between CGI header and data.
- So our completer Perl line looks like this:

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

Finally — Our complete script

Recall that our Perl CGI script must output the header and HTML code and must begin with a special first line.

Our complete first program (with nice comments) is a follows:

```
#!/usr/bin/perl
# hello.pl - My first Perl CGI program

print "Content-Type: text/html\n\n";
# Note there is a newline between
# this header and Data

# Simple HTML code follows

print "<html> <head>\n";
print "<title>Hello, world!</title>";
print "</head>\n";
print "<body>\n";
print "<h1>Hello, world!</h1>\n";
print "</body> </html>\n";
```

Writing, Creating and Running CGI Perl Scripts

We now know what a (simple) Perl script looks like.

Let us now look at how we create and run Perl scripts.

We will look at how we create Perl Scripts on a Macintosh or UNIX and how we run Perl scripts as standalone and CGI scripts on Macintosh and UNIX.

Writing/Creating Perl Scripts

Perl Scripts are basically text files with special perl syntax embedded in the text.

Therefore any **text editor** can be used to create and edit you Perl files.

On the Macintosh Computer **BBEdit Lite** is the recommended text editor.

Running Perl on Mac OS X/UNIX/LINUX Command Line

- Simply fire up a terminal window or
- Open Telnet connection to UNIX machine
- Make sure the Perl script is executable
  - The UNIX command:

    ```
    chmod +x myperl.pl
    ```

    achieves this.
  - To see whether a file is **execuable** use UNIX command `ls -l`,

`ls -l` To See if File is Executable Example

E.G.:

```
ls -l myperl.pl
```

You should see something like:

```
-rwxr-xr-x 1 dave staff 356 Nov 19 2003 myperl.pl
```

Look for the `x` in the User, Group and/or All file permissions

- Either simply type the file name from the command:

  `myperl.pl` , or
- Run the perl interpreter, `perl`, with the file name:

  ```
  perl myperl.pl
  ```

Test Perl Script Locally First

- If you run perl scripts from the command line they **DO NOT** function as a CGI script
- **However you can verify that the scripts syntax is correct**
  - and save wasted file copying to web server
- **Possibly you can verify that the scripts output is correct**
  - by manually viewing the script output on the command line
  - E.G. Basic HTML syntax
  - and save wasted file copying to web server

Running Perl on School's UNIX/LINUX Web Server

We assume that a Perl Script has been created and tested on a Macintosh Locally.

To run a CGI Perl script on UNIX, Simply:

- Samba File Copy or FTP (use **Fetch**) the Perl Script to the appropriate **cgi-bin** directory on UNIX (`project` or `public`).
- Put associated HTML file in appropriate **html** directory on UNIX (`project` or `public`).
- Reference Perl script either via
  - a FORM — Make sure URL is the Correct UNIX URL
  - Directly with a URL
- The URL is either `http://www.cs.cf.ac.uk/project/A.B.Surname/cgi-bin/file.pl`, or `http://www.cs.cf.ac.uk/user/A.B.Surname/cgi-bin/file.pl`.

Back
Close

CGI Script Input: Accepting Input To Perl Scripts

A CGI script will **typically require some form** of input in order to operate.

- In fact, only very trivial CGI scripts can be created without input.

We have introduced HTML Forms as a prime means of CGI input.

- However, there are several other forms of input to a CGI script.

In this section we will study:

- What form of input a CGI can receive
- How a CGI receives input.
- How to process the input in a CGI Perl script
- How a useful Perl library makes this (and other) tasks easy.

Back
Close

Accepting Input from the Browser

A CGI script can receive data in one of four ways:

Environment Variables — It gets various information about the browser, the server and the CGI script itself through specially named variables automatically created and setup by the server. More on these later.

Standard Input — Data can be passed as standard input to CGI script. Usually this is through the `POST` method of an HTML Form. (Standalone Perl scripts get standard input from the keyboard or a file.)

Back
Close

Accepting Input from the Browser (Cont.)

Arguments of the CGI Script — If you call a CGI script directly or use the `GET` method of HTML Form posting information is passed as arguments of the CGI script URL.

- Arguments are followed a `?` after the CGI script URL and multiple arguments are separated by `&`.
- For example:
  `http://host/cgi-bin?arg1&arg2`
  The arguments are usually in the form of name/value pairs (More Soon).

Back
Close

Accepting Input from the Browser (Cont.)

Path Information — Files which may be read by a CGI script can be passed to a CGI script by appending the file path name to the end of the URL **but** before the `?` and any arguments.

For example:

`http://host/cgi-bin/script/mypath/cgiinput?arg1&arg2`

Path information is useful if a CGI script requires data that

- Does not frequently change,
- Requires a lot of arguments and/or
- Does not rely on user input values.
- Path Information often refers to files on the Web server such a configuration files, temporary files or data files.

Passing Data to a CGI Script

As mentioned above there are a few ways to pass data to a CGI script.

- Path Information and Arguments of a CGI Script call are **explicit** — you (or the HTML Form) have to actually specify the information ad part of the call.
- Standard input and Environment variables are **more transparent** — we will need to deal with accepting the input within the CGI script.

Let's consider the arguments of a CGI script call further for the moment.

The `GET` method of Form posting or a **direct URL** call to a CGI script may use this method.

NOTE: Using the direct method of CGI Script call is a good way to debug possible Form/CGI script interaction problems.

CGI Conventions

There are several conventions adopted when passing arguments to a CGI script:

- Different fields (*e.g.* name value pairs are separated by an **ampersand (&)**.
- Name/value pair assignments are denoted by an **equals sign (=)**.
  - The format is `name=value`.
- Blank spaces must be denoted by a **plus sign (+)**.
- Some special characters will be replaced by a **percent sign (%)** followed by a **2 digit hexadecimal (ASCII Value) code**.
  - For example if you need to input an **actual &, %** or **=** character as input.

The `GET` Form posting method does these things automatically.
Note: You may need to construct these things yourself if call the CGI script direct from a URL.

A Simple Form CGI Script Call

Let us now return to our minimal form example introduced previously and examine how the input is passed to a CGI script. We will then examine how the actual CGI script receives and processes the input data.

Recall the form simply has a single Text entry field and a submit button:

Figure 27: The Minimal Form

If we set the Form method attribute to `GET` via:

```
<form method = "get" action = "http://.../cgi-bin/minimal.cgi">
<input type="submit">Data: <input name="myfield">
</form>
```

CGI Input via a URL

If you enter some data in the Text field and click on submit then the call to the CGI script looks something like

`http://myhost/minimal.pl?myfield=dddd`

where `minimal.pl` is the CGI script *actioned* by the form,

If you want to call the CGI script yourself (by-passing the Form) you simply mimic to input above.

Try typing:

```
http://www.cs.cf.ac.uk/user/
Dave.Marshall/cgi-bin/minimal.pl?myfield=mydata
```

in the browser location bar.

Exercise: Change to Form method attribute to `POST` and observe the difference in the CGI call.

The Other Side of CGI — Receiving and processing information in a CGI (Perl) script

There are basic ways to process or parse input in a Perl

- Do it yourself — write several lines of Perl code to process the input.
- Use **pre-written Perl libraries** — somebody has already done the arduous task of writing Perl code to parse input.

Which option would you choose?

Which option would you choose?

Bear in mind that:

- Input can be provided by different mechanism.
- Input of many arguments, name/value pairs may get complex.
- We do not know enough Perl to do it ourselves yet!!
- Prewritten code has been extensively tested — It should work.

THE INFORMED VIEW IS TO USE: pre-written Perl libraries

A Simple Perl CGI Input Library

There are many Perl libraries available to read and parse CGI input.

- These are freely available on theWorld Wide Web via the Comprehensive Perl Archive Network (CPAN)

The library that became one of first the standard CGI libraries is the **cgi-lib.pl** library.
Further Information on this library is available from http://cgi-lib.berkeley.edu/.

Copies of the actual Perl file are available from above web site and locally.

You should find copies of the `cgi-lib.pl` file in:

- You can download the file from http://cgi-lib.berkeley.edu/ or (Locally) http://www.cs.cf.ac.uk/User/Dave.Marshall/cgi-lib.pl.

CGI.pm — A more complete, fully featured and advanced Perl library

Note: A more complete, fully featured and advanced Perl library CGI.pm exists

But this is beyond our current Perl knowledge.

Please check this out later (for projects etc.)

The `cgi-lib.pl` library

The `cgi-lib.pl` Perl library simply consists of handy, easy-to-use Perl functions. The library is more than simply a means of processing CGI input. The library includes subroutines to:

- Read and parse CGI input — a value(s) for a given name can be easily found.
- Conveniently format CGI output.
  - Conveniently return Headers and Bottoms of standard CGI output.
  - Conveniently return URLs.
  - Conveniently return CGI Error Codes.
- Print in HTML format all name/value pairs input.
- Print in HTML format Environment variables.

CGI Input via `cgi-lib.pl` library is Easy

The `cgi-lib.pl` input routines can accept all and process all methods of input (*e.g.* `GET` and `POST` methods).

You do not have to worry about which mechanism has been adopted by HTML Form

Let us now develop a `minimal.pl` CGI routine that accepts input form our minimal form and sends back HTML that echoes the input data.

We will use the `cgi-lib.pl`.

A Minimal Form Response CGI Perl Script

In this subsection we will develop a `minimal.pl` CGI routine that accepts input form our minimal form and sends back HTML that echoes the input data.

We will use the `cgi-lib.pl` to

- Parse the input from the form.
- Format the HTML output.

We will need to learn some more basic Perl:

- How to include and call Perl libraries.
- How to call Perl subroutines

Our `minimal.pl` Script

The first thing our Perl script will need to do is to include the Perl library file `cgi-lib.pl`.

The Perl command `require` will load in any external Perl file.

- It is easier and sometimes essential that all library files exist in the same folder or directory as the main Perl script calling the library.

Therefore make sure that all Perl files required for a Perl program do exist at the same folder or directory level.

Thus to include our `cgi-lib.pl` file we need the Perl command:

```
require "cgi-lib.pl";
```

Note the format of the `require` command has the Perl file listed in ``...''.

Back
Close

Our `minimal.pl` Script (Cont.)

Having included the library we can call on its many useful subroutines.

The `&ReadParse()` subroutine reads either `GET` or `POST` input and conveniently stores the name/value pairs in a Perl array (We will meet these formally shortly for now we simply use the array).

Thus a Perl call of the form:

```
&ReadParse(*input);
```

will store the input in an array `input`.

& is used to indicate a Perl subroutine call.

Back
Close

Our `minimal.pl` Script (Cont.)

Next we will need to extract out the relevant value of a given name.

This is relatively simple. Perl is very good a process data of this kind.

In our current example there is only one input field and we are therefore only interested in the value associated with the `myfield` name.
To get this value you simply do:

```
$input{'myfield'}
```

Thus to print out the value typed we could do something like:

```
print "You typed: " . $input{'myfield'} . "\n";
```

Back
Close

Our first Complete minimal Perl script

So pulling together all we have learnt so far. A Perl script to take our minimal form input and return in HTML the value type could be:

```
#!/usr/bin/perl# minimal.cgi
# This is the minimalist form script
# to demonstrate the use of
# the cgi-lib.pl library

require "cgi-lib.pl";

# Read in all the variables set by the form

&ReadParse(*input);

print "Content-Type: text/html\n\n";
print "<html> <head>\n";
print "<title>Minimal Input</title>\n";
print "</head>\n";
print "<body>\n";

print "You typed: " . $input{'myfield'} . "\n";

print "</body> </html>\n";
```

Back
Close

A second minimal Perl script

We can further than this and exploit some more `cgi-lib.pl` subroutines.

Nearly every CGI output has:

- Exactly the same header output.
- Similar HTML head information
- Similar HTML ending

Fortunately subroutines exist to save us typing this same information all the time.

The `&PrintHeader` subroutine returns the string:

```
Content-Type: text/html\n\n
```

Thus we can use `print` in conjunction to produce our CGI header output via:

```
print &PrintHeader;
```

The `&HtmlTop` subroutine

The `&HtmlTop` subroutine accepts a single string argument, `MY TITLE` say, and return an HTML Head and Body (opening only) with the argument as the HTML page `TITLE` and `H1` Header. *I.e.*

```
<html>
<head>
<title>MY TITLE</title>
</head>
<body>
<h1>MY TITLE</h1>
```

which is rather useful.

The `&HtmlBot` subroutine

The `&HtmlBot` subroutine is the compliment of `&HtmlTop` and returns:

```
</body>
</html>
```

Thus we can use these functions and we only need to provide the main HTML body ourselves.

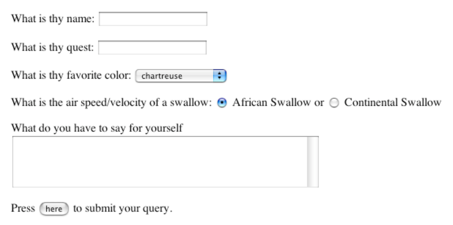
Thus we develop a better minimal.pl script as follows

A Better `minimal.pl` Script

```
#!/usr/bin/perl

# minimal.cgi
# This is the minimalist form script
# to demonstrate the use of
# the cgi-lib.pl library

require "cgi-lib.pl";

# Read in all the variables set by the form

&ReadParse(*input);

# Read in all the variables set by the form
  &ReadParse(*input);

# Print the header + html top
  print &PrintHeader;
  print &HtmlTop ("Minimal Input");

print "You typed: " . $input{'myfield'} . "\n";

print &HtmlBot;
```

Multiple argument input to a Perl CGI script

Let us now return to a more complex form example. We previously looked at the Python Quiz form.



What is thy name:

What is thy quest:

What is thy favorite color: chartreuse

What is the air speed/velocity of a swallow: ● African Swallow or ○ Continental Swallow

What do you have to say for yourself

Press here to submit your query.

Figure 28: Multi Field Form

The Python Quiz Form

The HTML to produce this is:

```
<H1>Python Quiz: </H1>

<form  method = "post" action = "simple-form.cgi">
What is thy name: <input name="name"><P>
What is thy quest: <input name="quest"><P>

What is thy favorite color:
 <select name="color">
 <option selected>chartreuse
 <option>azure
 <option>puce
 <option>cornflower
 <option>olive draub
 .....
 <option>amber
 <option>mustard
 </select>
<P>

What is the weight of a swallow: <input type="radio" name="swallow"
value="african" checked> African Swallow or
<input type="radio" name="swallow" value="continental"> Continental
Swallow
<P>

What do you have to say for yourself
<textarea name="text" rows=5 cols=60></textarea>
<P>

Press <input type="submit" value="here"> to submit your query.
</form>
```

The Python Quiz Form

Note: that the Method attribute is set to `POST`.

- This desirable since many name/value pairs are sent to the CGI script.

If the method was set to `G ET` instead the call would look something like this:

```
http://dave.cs.cf.ac.uk/.../python.pl?name=Dave&
quest=Find+Holy+Grail&
color=olive+draub&
swallow=continental&
text=I+am+Tired
```

Also:

- Note relevant character substiutions and argument dividors
- This is all one line of a URL!!

The Post method sends all this dat via standard input which is far neater.

Python Form CGI Input

Clearly fhe form input has several input name/value pairs: `name, quest, color, swallow` and text

To extract out the relevant value for a given name is straightforward though:

- Use `&ReadParse(*input)`
- Extract out the value for a given name using `$input{'name'}`, `$input{'quest'}`, etc.

Modifying Text Area Input

It is also useful to process the form input for a `Textarea` field so that line breaks are inserted

- Since HTML does not preserve linebreaks and carriage returns will be present in the multiline output.

The Perl commands:

```
($text = $input{'text'}) =~ s/\n/\n<BR>/g;
```

does this.

We will explain the Perl commands here in detail later and briefly now.

Text Area Modification Brief Explanation

This Perl is a little complex for complete study now. Essentially the following occurs

- The input value to the `text` name is copied to a `$text` array.

  `($text = $input{'text'})`

- All end of lines `\n` characters are substituted (`s/.../` command) by `\n<BR>`.

  `s/\n/\n<BR>/g`

  The `s/old_sting/new_string/` command is commonly used in Perl.

  The `g` at the end of the command indicates a *global* substitution (all instances get replaced) rather than the (default) first found.

Complete `python.pl` Script

So our complete Perl script is as follows:

- Input is read and parsed as usual with `cgi-lib.pl` routine `&ReadParse(*input)`
- Values pertaining to names (`name, color, quest, swallow` and `text` are sought.
- The `text` value is processed as above.
- The values are printed out in HTML format.
- `cgi-lib.pl` subroutines are used to output CGI Header, and HTML Top and bottom.
- Another `cgi-lib.pl` subroutine, `&PrintVariables(*input)`, is also used to print out all the input name/value pairs.

The complete Perl code is as follows:

```
#!/usr/bin/perl

require "cgi-lib.pl";

# Read in all the variables from form
&ReadParse(*input);

# Print the header
print &PrintHeader;
print &HtmlTop ("cgi-lib.pl demo form output");

# Do some processing, and print some output
# add <BR>'s after carriage returns
# to multline input, since HTML does not
# preserve line breaks
($text = $input{'text'}) =~ s/\n/\n<BR>/g;

 print << ENDOFTEXT;

You, $input{'name'}, whose favorite color is $input{'color'} are on a
quest which is $input{'quest'}, and are looking for the air speed/velocity of an
$input{'swallow'} swallow.  And this is what you have to say for
yourself:<P> $text<P>

ENDOFTEXT

# If you want, just print out a list of all of the variables.
print "<HR>And here is a list of the variables you entered...<P>";
print &PrintVariables(*input);

# Close the document cleanly.
print &HtmlBot;
```

Basic Perl Programming

In this section we will explore some basic Perl programming concepts.

Many of these concepts are similar to what you have learned in **Java** or other programming languages.

However there **quite a few differences**.

Here we will focus on how Perl

- Defines and uses variables
- How basic math and string operations are performed.
- Learn some very useful Perl functions

Perl Variables

Perl regards variables as being of one of three basic types:

Scalar — denoted by a `$` symbol prefix.

A scalar variable can be either a **number** or a **string**.

Array — denoted by a `@` symbol prefix.

Arrays are indexed by numbers.

Associative Array — denoted by a `%` symbol prefix.

Arrays are indexed by **strings**. You can look up items by **name**.

Note: This is quite different than Java.

- But Hopefully things are easier here once you get used to the **different syntax**.

Scalar Variables

Scalar variables can be either a number or a string

- What might seem confusing at first sight actually makes a lot of sense and can make programming a lot easier.
  - You can use variable types almost interchangeably.

    For Example: Numbers first then strings later
  - Numbers are numbers — there is no integer type **by default (These is an `integer` package but we do not worry about this in this course.**

    Perl regards all numbers as floating point numbers for calculations etc. **UNLESS OTHERWISE INSTRUCTED**

Defining Scalar Variables

You define scalar variables by assigning a value (number or string) to it.

- You do not declare variable types at a special place in the program as in C, PASCAL etc.
- It is a good idea to declare all variables together near the top of the program.

The following are simple examples of variable declarations:

```
$first_name = "David";
$last_name = "Marshall";

$number = 3;

$another_number = 1.25;

$sci_number = 7.25e25;

$octal_number = 0377; # same as 255 decimal

$hex_number = 0xff; # same as 255 decimal
```

Defining Scalar Variables — Perl Syntax Notes

```
$first_name = "David";
$last_name = "Marshall";

$number = 3;

$another_number = 1.25;

$sci_number = 7.25e25;

$octal_number = 0377; # same as 255 decimal

$hex_number = 0xff; # same as 255 decimal
```

NOTE:

- All references to scalar variables need a $.
- Perl commands end with a **semicolon (;)**. This can be omitted from last lines of "blocks" of statements.
- All standard number formats are supported integer, float and scientific literal values are supported.
- **Hexadecimal** and **Octal** number formats are supported by 0x and 0 prefix.

String Scalar Variables

Strings are a sequence of characters.

Perl has two types of string:

Single-quoted strings — denoted by '....'.

- All characters are regarded as being **literal** characters.
- That is to say special format characters like `\n` are regarded as being two characters \ and `n` with no special meaning.
- Two exceptions:
  - To get a single-quote character do `\n'`
  - To get a backslash character do `\\'`

Double-quoted strings — Special format characters \ have a distinct meaning.

Double-quoted strings — Special Format characters

Some special format characters include:

```
\n      newline
\r      carriage return
\t      tab
\b      backspace
\\      backslash character
\"      double-quote character
\l      lower case next letter
\L      lower case all letters until \E
\u      upper case next letter
\U      upper case all letters until \E
\E      Terminate \L or \E
```

Operators

Just as in most standard programming languages variables are operated on or assign results of operations.

In Perl scalar variables and constants can be mixed an assigned as normal.

Common arithmetic operation are denoted by `+,-,*,/,` `%` (modulus) and `**` (power).

For example:

```
$x = 3 + 1;
$y = 6 - $x;

$z = $x * $y;

$w = 2**3;  # 2 to the power of 3 = 8
```

String Operators

Strings can be concatenated by the . operator.

For example:

```
$first_name = "David";
$last_name = "Marshall";

$full_name = $first_name . " " . $last_name;
```

we need the " " to insert a space between the strings.

Strings can be repeated with x operator

For example:

```
$first_name = "David";

$david_cubed = $first_name x 3;
```

which gives "DavidDavidDavid".

String Operators (Cont.)

Strings can be referenced inside strings

For example:

```
$first_name = "David";

$str = "My name is: $first_name";
```

which gives

```
"My name is:  David".
```

Conversion between numbers and Strings

This is a useful facility in Perl:

- Scalar variables are converted automatically to string and number values according to context.

Thus you can do

```
$x = "40";
$y = "11";

$z = $x + $y;  # answer 51

$w = $x . $y;  # answer "4011"
```

Note if a string contains any trailing non-number characters they will be ignored.
I.e. " 123.45abc" would get converted to 123.45 for numeric work.
If no number is present or there are non-number characters first in a string it is converted to 0.

Binary Assignment Operators

In common with Java and C, Perl has two short hand operators that can prove useful.

In many statements we frequently write something like:

```
$a = $a + 1;
```

we can write this more compactly as:

```
$a += 1;.
```

This works for any operator so this is equivalent:

```
$a = $a * $b;
$a *= $b;
```

Auto Increment/Decrement Operators

You can also automatically increment and decrement variables in Perl with the ++ and -- operators.

For example all three expressions below achieve the same result — adding one to the value of a:

```
$a = $a + 1;
$a += 1;
++$a;
```

Prefix/Postfix Auto Increment/Decrement Operators

The ++ and -- operators can be used in **prefix** and **postfix** mode in expressions.

- There is **subtle** a difference in their use.

In **Prefix Mode**: the operator comes **before** the variable and this indicates that the value of the **operation** be used in the expression:

E.g.

```
$a = 3;
$b = ++$a;
```

results in a being **incremented to 4 before** this new value is assigned to b.

That is to say BOTH a and b have the value 4 after these statements have been executed.

Prefix/Postfix Auto Increment/Decrement Operators (Cont.)

In **Postfix Mode**: the operator comes **after** the variable and this indicates that the value of the **variable** before the operation be used in the expression and then the variable is incremented or decremented:

E.g.

```
$a = 3;
$b = $a++;
```

results in the **value** of a (3) being assigned to b and then a gets **incremented to 4**

That is to say that after these statements have been executed
b = 3 and a = 4.

The `chop()` operator

`chop()` is a useful operator which takes a single argument (within parenthesis) and simply removes the last character from the string. The new string is returned as a value to the expression.

Thus
`chop('suey')` would give the result `'sue'`

The `chomp()` operator

`chomp()` is a similar operator which takes a single argument (within parenthesis) and removes the last character from the string **only if it is the newline, `\n`, character**.

Why is `chomp()` useful?

- Most strings input in Perl will end with a `\n`:
  - Most lines of text and some strings read in by Perl will have this `\n` character present (**more on reading files etc. soon**)
- If we want to line/string operations for output formatting and many other processed then the `\n` might be inappropriate.
- `chomp()` can easily achieve this.

Arrays

What is an Array?

An array, in Perl, is an **ordered list of scalar data**.

- **This is quite different to standard arrays in JAVA or C**
- Syntax is also slightly different.
- Each element of an array is an separate scalar variable with a independent scalar value — unlike JAVA or C.
- Arrays can therefore have **mixed** elements, for example

  ```
  (1,"fred", 3.5)
  ```

  is perfectly valid.

Literal Arrays

Arrays can be defined literally in Perl code by

- Simply enclosing the array elements in **parentheses, (...)** , and
- **Separating each array element** with a **comma**.

For example

```
(1, 2, 3)
("fred", "albert")
() # empty array (zero elements)
(1..5) # shorthand for (1,2,3,4,5)
```

Indexed Arrays

You declare an **ordinary indexed array** by giving it:

- A **name AND**
- Prefixing it with a `@`

Values are assigned to arrays in the usual fashion:

```
@array = (1,2,3);

@copy_array = @array;

@one_element = 1;
# not an error but creates the array (1)
```

References to Arrays Within Arrays

Arrays can also be **referenced within** the literal array list, for example:

```
@array1 = (4,5,6);

@array2 = (1,2,3, @array1, 7,8);
```

This results in the elements of `array1` being **inserted** in the appropriate parts of the list.
Therefore after the above operations

```
@array2 = (1,2,3,4,5,6,7,8)
```

This means **Array cannot contain** other Arrayelements recursively **only scalars allowed as Array elements**

Indexing Array Elements

Elements of arrays are indexed by index, For Example:

```
$array1[1] = $array2[3];
```

Assigns "third" element of `array2` to "first" element of `array1`.

Each array element is a **scalar** so we reference each array element with

- The `$` prefix — to inidcate a scalar value being used
- Use `[n]` brackets for index to element `n`.

BIG WARNING:
Array indexing starts from **0** in Perl (like JAVA and C).
So

```
@array = (1,2,3,4,5,6,7,8);
```

The index `$array[0]` refers to 1 and `$array[5]` refers to 6.

Finding Array Length in Perl

Arrays are essentially dynamic lists in Perl:

- The length of an array can change in the program (**some useful dynamic functions next**)
- Knowing the the length at any given time is useful:

  For example: **To loop through array and print/process each array element**

If you assign a **scalar** to an **array name (plus `@` prefix)** the scalar gets assigned the length of the array, **E.g.**:

```
@array2 = (1,2,3,4,5,6,7,8);

$length = @array2; # length = 8
```

Some Useful Array Functions

`push()` and `pop()`

One common use of an array is as a **stack**.
`push()` and `pop()` **add** or **remove** an item from the right hand side of an array.

Example:

```
push(@mystack,$newvalue);
# add new value to stack

$off_value = pop(@mystack);
# take last element off array
```

shift() and unshift()

Like push() and pop() except put values on and take values off the left side of an array.
Example:

```
shift(@mystack,$newvalue);
# add new value to left of array

$off_value = unshift(@mystack);
# take last element off left of array
```

shift() and unshift(),push() and pop() may be used **together** to create more fancy data structures such as queues, double ended queues (deques) and others.

reverse()

As one would expect this will reverse the ordering of list.

For example:

```
@a = (1,2,3);

@b = reverse(@a);
```

results in b containing (3,2,1).

sort()

This is a useful function that sorts an array.

- **NOTE**: Sorting is done on the string values of each number **alphabetically** — Not **numerically**

For Example:

```
@x = sort("small","medium","large");

# gets @x = ("large","medium","small");

@y = sort(1,,32,16,4,2);

# gets @x = (1,16,2,32,4);
```

chop()

How does chop() (and other similar operators) work on arrays? Just as on a scalar string it removes the last character of every element from an array, for example:

```
@x = ("small","medium","large");

chop(@x);

# gets @x =  ("smal","mediu","larg")
```

Use pop() unshift() to remove entire array elements.

Hashes/Associative Arrays

Hashes or (Previously called) Associative arrays are a very useful and commonly used feature of Perl.

Hashes basically store tables of information where:

- The lookup is the right hand key (a string) to an associated scalar value.
- Again Hash scalar elements can be mixed "data types" values — as with ordinary arrays.

Perl Flashback: We have already been using Associative arrays for name/value pair input to CGI scripts.

Hashes are denoted by a % prefix.

- When you declare an Hash the key and associated values are listed in consecutive pairs.

Hash Array Example

Lets assume we have the following **"secret code"** lookup example:

name	code
dave	1234
peter	3456
andrew	6789

We would declare a Perl Hash to perform this lookup as follows:

```
%lookup = ("dave", 1234,
           "peter", 3456,
           "andrew", 6789);
```

To reference a particular value you would do, for example:

```
$lookup{"dave"}
```

Creating New Hash Table Elements

You can create new elements by assignments to new keys.

E.g.

```
$lookup{"adam"} = 3845;
```

After this operation a new table item (Hash lookup and value) will have been created.

You do new assignments to old key lookups also, for example:

```
# change dave's code
$lookup{"dave"} = 7634;
```

Hash Operators

`keys()`
The `keys(%arrayname)` lists all the key names in a specified Hash.

- The answer is returned as an **ordinary index array**.

E.g.

```
@names = keys(%lookup);
```

`values()` This operator returns the values of the specified Hash Table.

E.g.

```
@codes = keys(%lookup);
```

Deleting Hash Table Entries

delete() deletes an associated key and value by key reference.

E.g.

```
# scrub adam from code list
delete $lookup("adam");
```

Further Perl Programming

Statement Blocks

In Perl statement blocks are enclosed in pairs of **curly** brackets {....}:

```
{

   statement_1;
   statement_2;
   statement_3;

   .....

   statement_n;
}
```

If Statement

The if statement has a variety of forms.

The simplest is:

```
if (expression)
  {  true_statement_1;
     .....
     true_statement_n;
  }
```

which means that if expression is evaluated as being **true** then execute the block of statements.

In Perl,

- **False** is regarded as any expression which evaluates to 0 (zero).
- **True** any expression which is not false (non-zero).

If/Then/Else Statement

An else may be added to provide a block of statements to be executed upon a **false** evaluation of the expression:

```
if (expression)
  {  true_statement_1;
     .....
     true_statement_n;
  }
else
  {  false_statement_1;
     .....
     false_statement_n;
  }
```

Curly braces are required for each block even if **only one** statement is present.

If/Then/Else Example

For example:

```
$age = ; # whatever ??
if ($age < 18)
  { print "You cannot Vote or have a beer, yet.\n";
  }
else
  { print "Go and Vote and then have a beer.\n";
  }
```

Back Close

Unless Statement

There is an `unless` statement in Perl which can be regarded as the **negative** of `if` — `unless` states:

- If the control expression is **not** true, do ....

For example:

```
$age = ; # whatever ??
unless ($age < 18)

  { print "Go and Vote and then have a beer.\n";
  }
```

`unless` can have an `else`, too.

Back Close

Multiple Else Ifs — the `elsif` Statement

If you have more than one branch then the `elsif` can be added to the `if`.

- You cannot have an `else if` in Perl
- **You must use `elsif`.**
- Last conditional probably a lone `else`

Back Close

`elsif` Statement Example

For Example:

```
$age = ; # whatever ??
if ($age < 16)
  {
    print "Hi, Junior\n";
  }
elsif ($age < 17)
 {
    print You can ????\n";
  }
elsif ($age < 18)
 {
    print You can learn to drive\n";
  }
else
  { print "Go and Vote and then have a beer.\n";
  }
```

Note: The last `else`.

Back Close

Comparison operators for numbers and strings

Perl has different operators for comparing numbers and strings:

- Frequently used in `If/Unless/Elsif` conditional expressions
- Used in other computations also

They are defined as follows:

Comparison	Numeric	String
Equal	`==`	`eq`
Not Equal	`!=`	`ne`
Less than	`<`	`lt`
Greater than	`>`	`gt`
Less than or equal	`<=`	`le`
Greater than or equal	`<=`	`ge`

Note: Different operators for Numeric and String comparison

- Even though scalar variables can be of either type in a program
- Need to take care when performing comparison in some Perl programs

Logical operators

In controlling the logic of a conditional expression logical operators are frequently required.

In Perl,

- The logical AND operator is `&&` and
- The logical OR is `||`

Useful Example: Checking If A Valid Number Exists in a Variable

For example to check if a valid number exists in a variable `$var` you could do:

```
if ( ($var ne "0") && ($var == 0))
  { # $var is NOT a number
  }
else
  { # $var is a NUMBER
  }
```

How Example does this work?

- Recall: Perl evaluates any string to 0 if it is not a number:

- If `$var` is a regular number (or a string with leading digits), but not 0, then the string test (`$var ne "0"`) is FALSE as is the numeric test (`$var == 0`).
- If `$var` is a regular string (with no leading digits), but not "0", then the string test (`$var ne "0"`) is FALSE but the numeric test (`$var == 0`) is TRUE (since any string with nor leading digits is zero numerically).

  But the complete `AND`ed expression is still FALSE
- The only problem we have is if `$var` is 0 (number) or "0" (Single Character) — Perl regards these are identical in a variable.

  Here we need the (`$var ne "0"`) test which evaluates FALSE in this case and therefore makes the `AND`ed expression FALSE

The `for` statement

Just like in Java and C we have loops.
The simplest is the `for` loop.

- Actually behaves like JAVA/C's statement

The format of the `for` statement is:

```
for ( initialise_expr; test_expr; increment_expr )
  {
     statement(s);
  }
```

For example:

```
for ( $i = 1; $i <= 10; ++$i )
  { # count to 10
     print "$i\n";
  }
```

The `while` statement

The `while` statement is as follows:

```
while (expression)
  { # while expression is true execute this block

      statement(s);
  }
```

For example:

```
i = 1;
while (  $i <= 10 )
  { # count to 10
     print "$i\n";
     ++$i;
  }
```

The `until` statement

The `until` statement says **do while expression is false** as declared is as follows:

```
until (expression)
  { # until expression is false execute this block

      statement(s);
  }
```

For example:

```
i = 1;
until (  $i > 10 )
  { # count to 10
     print "$i\n";
     ++$i;
  }
```

The `foreach` statement

The `foreach` statement iterates through items in a list:

- No counterpart in JAVA or C.
- Similar Unix Shell commands exist.

The statement has the following format:

```
foreach $i (@some_list)
  {  # $i takes on each list item value in turn
     statement(s);
  }
```

A `foreach` Example

For example:

```
@a = (1, 2, 3, 4, 5);
foreach $i (reverse @a)
  { # reverse is a functiion that flips the list
    print $i;
  }
```

- **Note**: We use `reverse` to return a flipped list in the `foreach` list.