



INDIANA UNIVERSITY SOUTH BEND

PERL AND CGI

A Tutorial by Abby Buell
For the Department of Computer and Information Sciences
Y398 Internship

INSIDE THIS REPORT

Who It's For	1
What You Need	1
What Is Perl?	2
Getting Started	2
Hello, World	4
Variables	5
Control Structures	9
Subroutines	11
Perl References	13
Modules	14
System Interaction	15
Input From User	16
Working With Files	16
Data Manipulation	17
CGI	19
References	26
Appendix A	27
Appendix B	28

Who It's For

This is a tutorial directed at Indiana University students at the South Bend campus who would like to use [Perl](#) to write and publish [CGI](#) scripts on IUSB's web servers; the CS server is available to Computer Science students, and the Mypage server is available to all students at IUSB. Also, the Athos server is available for departments and clubs.

It's helpful to have some knowledge of a [Linux/ UNIX](#) environment, common programming concepts, and also some basic understanding of web publishing, [HTML](#), or [XML](#). HTML is not at all difficult to use, and you can find an excellent beginner's tutorial at:

<http://www.htmlgoodies.com/primers/html>

As for web publishing, all that basically means is that we're going to be sticking some html files in a folder on a web server. Simple enough.

Others (not IUSB students) will find most of the information in this tutorial helpful, but will need to disregard my references to specifics that are only relevant to IUSB's servers.

What You Need

This section is important in order to get started using Perl.

A Server with Perl

First and foremost, you need access to a server that has Perl available on it. IUSB students have this available to them. Others will need to have an account with a web hosting company that won't mind you playing around with Perl and CGI. I recommend finding a Linux/Unix server. Perl is available on Windows Server 2003, but its use in a Windows environment is somewhat different.

Alternatively, if you wanted to install Perl on your own system, it is completely open source and free, and available for download at <http://www.perl.com>.

Also, the most popular web server on the Internet, Apache, is completely free from <http://www.apache.org>. You can install it yourself if you have a C compiler. It's available for Windows platforms as well.

SSH

Computers on the IUSB campus have a program called SSH installed. This is just a secure shell client (similar to a telnet program) that will allow you to connect to IUSB's servers from any computer with internet access. You will most likely want to download a copy for your home computer or laptop so that you can log onto the server and work with Perl wherever you are. It's available for free from <http://iuware.iu.edu>. Go to this web page, and in the links at the left, click on 'Communication' and then click on Secure Shell (SSH). Log in with your user name and password and follow the instructions for downloading and installing the program.

CGIWrap

Those who want to use the Mypage server to publish web pages that use CGI are required by IUSB to use a program called CGIWrap. This does not require downloading and is really very simple to use. More about this when we're ready to use our first CGI script.

For now, if you'd like to read more about the program, this page has some additional information and FAQs:

<http://mypage.iusb.edu/cgiwrap/>

Text Editor

In order to write Perl code, you'll need a text editor. This could be Notepad for Windows, or emacs or vi with Linux. It's important not to use a word processor like Microsoft Wordpad or Word, because these programs save your files with additional formatting. Emacs and vi are available for use on the Mypage and CS servers; you don't need to download them.

What is Perl?

Perl stands for Practical Extraction and Reporting Language. You may recognize many features in Perl that are similar to other languages such as C, shell scripting, and Lisp. It is an open source server side general purpose programming language, and is a language specifically designed for processing text, and because of this trait is one of the most popular languages for writing Common Gateway Interface (CGI) scripts. When you submit a complex form from your browser window, for example, a Perl script may handle the processing of the information. Perl scripts are not embedded within HTML pages and do not download to the web browser but reside on the server. They execute by being triggered from commands within HTML pages or other scripts and may produce HTML output that does download to the web browser.

The current version of Perl is 5.8. An experimental version, Perl 5.9.4, is available for download from <http://www.perl.com>. Version 6.0 is also in the works, as Perl becomes more purely object-oriented. According to the Perl website:

“The fundamental syntax is still the same. It's just a little cleaner and a little more consistent. The basic feature set is still the same. It adds some powerful features that will probably change the way we code in Perl, but they aren't required.”

Currently, Perl version 5.8 is on both the Mypage and CS servers. On other servers, to tell what version of Perl you're using on Linux/UNIX, type

```
perl -v
```

on the command line. This will tell you what version you're running, as well as where to find complete Perl documentation, if you're interested.

Getting Started

If you're going to log onto the CS server from one of the Linux labs on campus, simply open a terminal window.

Otherwise, open your SSH program and click on “Quick Connect” and log on to one of the following:

For the Mypage Server

Host Name: mypage.iusb.edu

User Name: Enter the user name that you use to log onto the ADS.
(Hit Enter or click “Connect”)

Password: Enter the password that you use to log onto the ADS.

For the CS Server (Computer Science students)

Host Name: cs##.iusb.edu (Note: replace ## with a number 01 – 32)

User Name: Enter the user name that you use to log into the linux network.
(Hit Enter or click “Connect”)

Password: Enter the password that you use to log into the linux network.

For the Athos Server (Departments and Clubs)*

Host Name: athos.iusb.edu

User Name: This is a user name for authorized departments or clubs.
(Hit Enter or click “Connect”)

Password: This is a password for authorized departments or clubs.

If you have any trouble logging in, first check your user name and password. If you’re sure they’re correct...

and you’re having trouble logging onto mypage.iusb.edu, you can call the campus Helpdesk at (574) 520-5555.

and you’re having trouble logging onto the CS server, try several different numbers in the hostname. If this doesn’t work, contact the system administrator.

There is additional information for the Mypage and Athos servers, as well as a little troubleshooting advice, at:

<http://www.iusb.edu/~sbit/web-publishing.shtml>

*Note: I won’t specifically mention the Athos server again. Publishing to the Athos server is primarily like publishing to the Mypage server, and further information can be found at the link above.

Also, I won’t include much troubleshooting information in this tutorial, but Appendix B can point you in the right direction.

Creating an html directory

For the Mypage Server:

At the prompt, type “spinweb” and hit enter. This creates a directory for you called html and sets your permissions appropriately so that you can publish web pages inside of that directory. It will ask you if you want a link to your homepage listed at <http://mypage.iusb.edu>; this is a personal choice and has no effect on the results of this tutorial. It also created a default homepage for you called index.html.

Now move into the newly created directory by typing cd html (typing the first few letters of public_html and hitting the tab key will complete the directory name for you) and hit enter.

For the CS server:

There is already a simple and concise document containing instructions for setting up a web pages directory on the cs server; therefore I won't repeat the instructions here. They can be found at:

http://www.cs.iusb.edu/labs/howto/howto_website.html

After following the instructions in that document, you should have a directory called `public_html` and should all of your permissions set correctly.

Creating the cgi-bin

You should presently be inside of your html directory. To check you can use the command `pwd` which displays the path of your working directory. If you're not in your html directory, move into it by using the `cd` command. Now, type the commands:

```
mkdir cgi-bin
chmod o+x cgi-bin
```

The first command creates the `cgi-bin` directory and the second command makes the directory world executable. This is necessary so that if you create a web page that calls a Perl script, the script will execute correctly for anyone who views that web page.

Now, all web pages that you create will need to be placed into the `html` (for Mypage) or `public_html` (for cs) directory, and all Perl programs will go into the `cgi-bin` directory.

Move into your `cgi-bin` by typing `cd cgi - bin` and keep the SSH program open so that you are able to experiment with Perl.

Hello, World

In Perl, the canonical "Hello, World" program looks like this:

```
#!/user/bin/perl
print "Hello, World!\n";      # Using the print function to
                              # display "Hello, World!"
```

The first line, `#!/user/bin/perl`, tells the program where to find Perl. It must be correct for your server. On both Mypage and cs servers, Perl is located at `usr/bin/perl`. This is typical for most servers. On other systems, the simplest way to ensure you have the correct path to Perl is to type:

```
whereis perl
```

The second line uses the `print` function to print the string literal, "Hello, World!" The `\n` is a newline character and doesn't appear in the printed string. The `#` sign designates a comment and is ignored when the program is being compiled.

You should be in your `cgi-bin` directory at this point. To create this Perl program, you can open up a text editor, Emacs for example, by typing on the command line

```
emacs hello_world.pl
```

This will create a file called `hello_world` with the extension `.pl`, which is used on IUSB's servers. Some servers may require the extension `.cgi`, or allow either extension. On any Linux/UNIX system, you also have the option of leaving off the extension.

Copy the program code from the screen and paste it into the text editor (in the SSH program, go to Edit, paste).

Save the file. In emacs, hit `ctrl+x`, followed by `ctrl+s`. If you're on a remote connection, close the file by hitting the keys (in emacs) `ctrl+x` followed by `ctrl+c`.

Set correct permissions on the file by using the command

```
chmod o+x hello_world.pl
```

This makes the file world executable. Now, you can execute it in several ways, by typing one of the following:

```
hello_world.pl  
perl hello_world.pl  
./hello_world.pl
```

It just depends on the server which will work and which will not. And note that the third command will only work if you're currently in the directory from which you're trying to execute the program. Otherwise, you can execute it from anywhere, but will need to type out the entire path name, for instance:

```
/home/abuel1/public_html/cgi-bin/hello_world.pl
```

Assuming that one of these commands works, you should see the words **Hello, World!** printed on the screen. If you receive an error, check to make sure that the path to Perl in the first line is correct, and check to make sure you didn't make a typo in the program code.

Variables

There are three main types of variables in Perl: scalars, Arrays, and hashes.

When declaring a new variable of any kind, the default is that it will be a global variable, which is usually not what you want. To make the variable local to the block of code in which it is declared in, use the keyword "my" preceding the declaration. This is considered good practice so that in a large program, you don't accidentally give a new variable a name that you've already used, or otherwise inadvertently change the value of a variable; this could cause problems and be difficult to debug.

On the following page is a table of some of the most commonly used operators in Perl.

Numeric Operator	String Operator	Function	Examples
+		Addition	<code>\$i + 1</code>
-		Subtraction, Negative Numbers, Unary Negation	<code>\$y - \$x</code>
*		Multiplication	<code>\$j * 2</code>
/		Division	<code>\$k / 2</code>
%		Modulus	<code>24 % 5</code>
**		Exponent	<code>2 ** 3</code>
==	eq	Comparison of equality	<code>\$i == \$j</code> <code>\$var eq "First"</code>
>	gt	Greater than	<code>\$i > 2</code> <code>\$var gt "c"</code>
<	lt	Less than	<code>\$i < 0</code> <code>\$var lt "a"</code>
<=	le	Less than or equal	<code>\$i <= 0</code> <code>\$var le "a"</code>
>=	ge	Greater than or equal	<code>\$i >= 2</code> <code>\$var ge "c"</code>
&&	and	True if both variables are true	<code>\$i && \$j</code> <code>\$code and \$name</code>
	or	True if one or both variables are true	<code>\$i \$j</code> <code>\$code or \$ID</code>
!	not	Unary operator; true if the variable is not true	<code>! \$m</code> <code>not \$m</code>
++		Increment operator	<code>\$i++</code>
--		Decrement operator	<code>\$i--</code>

Scalars

Scalars are single values. They are designated by a "\$" sign, and can be any kind of single value; a number, a string, or an object. In Perl, it is not even necessary to declare or initialize a scalar variable in any way before you can use it, however, you may want to do so anyway as part of good programming practice.

Here are some examples of declaring, initializing, and assigning values to scalars:

```
my $real_var1 = 3.45;
my $mystring = "This is a string";
my $temp = 49;
my $celsius = 5/9 * ($temp - 32);
```

You might notice that other than the “\$” sign that tells Perl you are working with a scalar value, Perl doesn’t care what kind of scalar it is that you are declaring. You don’t have to make any indication that something is a string, real number, integer, etc. Some say this makes Perl a “weakly typed” language. What happens is that when you want to do something with a variable, Perl evaluates it based on the context. For instance, it’s perfectly reasonable to have the following code in your program:

```
my $course1 = "3 credits";
my $course2 = "4 credits";
my $total_credits = $course1 + $course2;
print "There are $total_credits total semester credits.\n";
```

Perl adds the strings together based on context. The program will print out:

```
There are 7 total semester credits.
```

Arrays

Arrays are also sometimes called lists, and are designated by the “@” sign. Here are a couple examples of declaring an array (again using the “my” keyword to make sure that they are local variables):

```
my @districts = ("New York", "Chicago", "San Francisco",
                "Dallas");

my @numbers = (12..56);
```

In the second example, the two periods are called a range operator. The array now contains all numbers 12 through 56, inclusive.

The entire array can be then printed with the statement

```
print "@districts\n";
print "@numbers\n";
```

An array in double quotes is interpolated, and its elements are returned separated by spaces. The statement would still work without the quotes, but would print the elements as one long string with no spaces.

Also, an individual element of the array can be accessed like so:

```
print $districts[2];           # prints "San Francisco"
```

There are a couple of things here to take notice of; first, that the command to print the element with subscript 2 prints the third element of the array. As with many languages, subscripts in Perl begin with 0, not one. Second, notice that in this case, since we are dealing with one single element of the array, which is a scalar, we use the “\$” sign instead of “@” when working with an element.

Modifying array data

There are a couple of ways in which we could add more elements to the array. We

can use subscripting, or we can use the **push** function. For example, to add an element to the back:

```
$districts[4] = "Minneapolis";
push (@districts, "Minneapolis");
```

There are several more useful functions available for adding/removing data from arrays; they are:

```
shift      removes and returns the front element
pop       removes and returns the last element
unshift  adds to the beginning
```

Hashes

Hashes are also called associative arrays. They are designated by the “%” sign. In a hash, data *values* are stored with an associated *key*.

```
%email = ("John Burns", "jdburns\@iusb.edu",
          "Can Nguyen", "cnnguyen\@iusb.edu",
          "Emily Bronte", "embronte\@iusb.edu",
          "Christine E. Mendoza", "cemendoza\@iusb.edu", );
```

In the above example, The names are the keys and e-mail addresses are the values. The backslash “\” used in the strings containing the e-mail addresses is necessary because “@” is a special character, and in this case we just want to print the actual “@” sign. The backslash is an escape character and lets Perl know that it shouldn’t look for any special meaning as it normally would; just print the literal “@” as it appears.

To print out the elements in a hash is not quite as simple as printing out elements of a simple array; in this case, we need to loop through the hash with the **foreach** function.

```
foreach $key(keys %email)
{
    print "$key can be contacted at $email{$key}\n";
}
```

This will print out:

```
Can Nguyen can be contacted at cnnguyen@iusb.edu
John Burns can be contacted at jdburns@iusb.edu
Christine E. Mendoza can be contacted at cemendoza@iusb.edu
Emily Bronte can be contacted at embronte@iusb.edu
```

What’s happening

Working with hashes can be a little confusing at first. The code **\$key(keys %email)** is taking each key found in the hash and assigning it to the variable **\$key** for each iteration.

Here, **\$key** is just a variable we are creating simply by using it.

foreach behaves just like it sounds; for each key-value pair in the hash, we

execute whatever statements are found inside the curly braces (more on this in the section on Control Structures), in this case, the `print` statement.

In each iteration, the `print` statement prints the value held in `$key` at that time, and then prints the value that is associated with that key.

`$email{$key}` is how an individual element is referred to. If we were just interested in printing out one single data value of the hash, for instance, if we needed to know John Burn's email address, we could do it like this:

```
print "John's email address is $email{'John
Burns'}\n";
```

Which prints out:

```
John's email address is jdburns@iusb.edu
```

Modifying Hash Data

To remove a single hash key, use the Perl `delete` function:

```
delete $email{'John Burns'};
```

To delete the entire array, reinitialize the hash to an empty list:

```
%email = ();
```

To add a key/value pair to the `%email` hash, use a statement like this:

```
$email{'New Guy'} = "newguy\@iusb.edu";
```

Control Structures

Loops and conditional statements are a way to control the flow of a program.

Blocks

Blocks are a way to group statements in Perl. Blocks of code are surrounded by curly braces, as we saw here:

```
foreach $key(keys %email)
{
    print "$key can be contacted at $email{$key}\n";
}
```

In some languages such as C/C++, the curly braces would not be necessary in this case because there's only one line of code. This isn't the case in Perl, and you'll get an error if you neglect to include the braces around the print statement.

Blocks can also be nested inside other blocks.

The if Statement

The `if` statement is pretty straightforward. It's used to test a conditional statement for true or false. If true, the block of code following the statement is executed, and if false, it is just skipped. Here is an example:

```
if ($i == 9)
{
    print "The value of $i is 9.\n";
}
```

The `else` statement can be used to execute an alternative code block, like this:

```
if ($i == 9)
{
    print "The value of $i is 9. \n";
}

else
{
    print "The value of i is not 9\n";
}
```

while Loop

The `while` statement also tests a conditional statement and evaluates it to true or false. While true, the following code block continues to be executed until the conditional statement becomes false. Inside the code block, the variables used in the conditional statement must be modified in some way so that your program doesn't get stuck in an endless loop.

```
$i = 0;

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

The above code prints out each integer from 1 to 99, at which point the statement (`$i < 100`) becomes false and the loop ends.

for Loop

Alternatively, we could accomplish the same thing as the while loop above with a for loop. It would look like this:

```
for ($i =0 ; $i < 100; $i++)
{
    print "$i ";
}
```

This code prints the same result as the `while` loop above.

Subroutines

In Perl, user-defined functions are called subroutines. Subroutines are designated by the keyword **sub**.

Subroutines can have arguments passed to them. They can also return a value. All passed arguments, by default, are stored in a special array variable called `@_`. Elements in `@_` can be printed out together or accessed individually like any other array. For clarity's sake and to make the arguments easier to work with, you'll probably want to give the arguments meaningful variable names.

In Perl it's not necessary to explicitly return a value with the **return** statement. The last evaluated value will be automatically returned unless you specify otherwise, but even if this is what you want, using the **return** statement may still be preferable for clarity.

Here is an example of a subroutine that finds the greatest of the arguments sent to it.

```
sub greatest
{
  my $total_args = @_; #array evaluated in scalar context
  my $larger=@_[0];

  foreach (@_)
  {
    if ($_ > $larger)
    {
      $larger = $_;
    }
  }
  return $larger;
}

$largest_arg = greatest(3, 235, 45, 924);
print "$largest_arg\n";
```

There are a couple of things to note in preceding bit of text:

- We haven't seen **foreach** before. It's a looping function for use with arrays. It's a simple way to visit each element in an array.
- For each iteration of the loop, the current array element that we're looking at is held in the special variable `$_`. Values will always default to `$_` if you don't explicitly assign them to a variable.

We could have used this statement instead:

```
foreach $element(@_)
```

and this way each time through the loop when we look at an element, it's assigned to **\$element**

rather than \$._.

Within the program, the subroutine can be called in one of two ways:

```
greatest();    # Can be used if the subroutine has been declared
                # (This is how it's called in the code above)

&greatest();  # Acceptable anywhere
```

I mentioned before that you may want to assign elements in @_ to variable names. Perl allows this to be done in one single statement. Say, for instance, your subroutine receives five scalar arguments, and you want to assign the first two to different scalars, and the rest are to be placed in an array. You could accomplish this with the following line in your subroutine:

```
sub print_arguments
{
    ($first_arg, $second_arg, @the_rest) = (@_);
    print "$first_arg, $second_arg, @the_rest\n";
}
```

Here is how the above subroutine would be called:

```
print_arguments($arg1, $arg2, $arg3, $arg4, $arg5);
```

You could, in fact, pass this subroutine any number of arguments. The first two would always be assigned to \$first_arg and \$second_arg, respectively, and the rest would always go into @the_rest.

Subroutine arguments can be arrays or hashes as well. They are passed just like a scalar. If you are passing scalars and/or arrays and/or hashes, be careful not to make the following mistake:

```
sub print_arguments
{
    (@the_rest $first_arg, $second_arg) = (@_);
    print "@the_rest, $first_arg, $second_arg \n";
}
```

If you intended the first three elements to go into @the_rest and the latter two elements to go into \$first_arg and \$second_arg respectively, this won't work. All of the elements will go into @the_rest. Similarly, if you pass two or more arrays/hashes, this won't work:

```
sub print_arguments
{
    (@first_array, @second_array) = (@_);
    print "$first_arg, $second_arg, @the_rest\n";
}

print_arguments(@array1, @array2);
```

All of the elements of both @array1 and @array2 would go into @first_array, and @second_array would have no elements. The only way to effectively pass two or more arrays or hashes to a subroutine is to use references, which are the topic of the next section.

Perl References

Many languages have some concept of references. It's basically a piece of data that's used to refer to another piece of data. In C/C++, they're called pointers. Multiple references can point to the same piece of data. References know the type of data they point to and its location.

When you create a scalar like this:

```
$district = "New York";
```

A piece of memory is reserved for that variable, which you can envision as looking like this:

```
$district  
└──┬──  
└──┴──  
New York
```

And if you were to make a copy of that variable, like this:

```
$district_copy = $district;
```

You'd have two independent pieces of data, like this:

```
$district_copy           $district  
└──┬──                 └──┬──  
└──┴──                 └──┴──  
New York               New York
```

But in some cases you may want both variables to refer to the same piece of data. That's done by doing an assignment as you normally would, but putting a backslash in front of the variable that you want to create a reference to, like this:

```
$district_ref = \ $district;
```

Now the reference and the original piece of data look like this:

```
$district_ref ───▶ $district  
                  └──┬──  
                  └──┴──  
                  New York
```

Now the data stored in the variable `$district` can be accessed and/or modified through the variable `$district_ref`, by dereferencing it with another backslash, like this:

```
$$district_ref = Dallas;
```

Now the actual value stored in `$district` has been changed to "Dallas".

References to arrays and hashes, respectively, are created like this:

```
$array_ref = \@array;  
$hash_ref = \%hash;
```

Anonymous Storage

Anonymous storage is a concept related to references. In Perl, references to variables can be used even when the variable itself is out of scope. Once you have the reference, you may no longer need to original data. In fact, with Perl you can even create a reference to some data without first creating a variable that stores that data. This allows you to basically eliminate a step.

Here is how you could create a reference to a hash without first creating a hash variable:

```
$hash_ref = { "Emily Bronte", "embronte\@i usb. edu",
             "Christine E. Mendoza", "cemendoza\@i usb. edu"};
```

The curly braces enclose the hash and return a reference to it. This is called an anonymous hash.

An anonymous array can be constructed using brackets:

```
$array_ref = [ ("New York", "Dallas", "San Francisco",
               "Minneapolis") ]
```

References to lists like the one above are often stored in an array in order to create a structure that is functionally like a two-dimensional array.

Modules

Modules are routines that extend the capabilities of Perl. Perl comes with a standard distribution of modules, and many more pre-written and tested modules are available for download at <http://www.cpan.org> free of cost. However, only a system administrator can install a new module on the Mypage or CS servers. It's possible that this could be done if requested for educational purposes.

A module is included in a program by using the `use` keyword along with the name of the module. You can then use functionality from the module as if its code had been in your program. Here is an example of using the `Math::BigInt` module, which allows you to work with very large integers:

```
#!/user/bin/perl

use Math::BigInt;           #for use with large integers

my $x = Math::BigInt->new('127392421535261424659');
print $x * $x;

#prints 16228829064617738796943011605982333266281
```

In the line that looks like this:

```
my $x = Math::BigInt->new('127392421535261424659');
```

The new keyword is necessary to create a new instance using the `Math::BigInt` module. It could alternatively be written like this:

```
my $x = new Math::BigInt('127392421535261424659');
```

The `::` is called a scope resolution operator. It tells the Perl interpreter where to look for the module. `Math` is a module that contains different functions. `Math::BigInt` tells the interpreter to look in the `Math` module and use the function `BigInt`.

As a side note, you might sometimes see the following lines in a Perl program:

```
use warnings;  
use strict;
```

These lines aren't referring to modules; they're basically instructions to the compiler. `use warnings` will provide you with expanded error messages during program development; afterwards it should be removed. `use strict` enforces some good programming practices involving variables; for example, you won't be able to declare a global variable unless you do so explicitly (which I won't cover here).

You can find out which modules are currently available to you by typing

```
perldoc perllocal
```

If there is a particular module you'd like to learn more about, you can read its documentation by typing

```
perldoc module name
```

System Interaction

It is often useful to interact with Linux when running a program. Perl makes it very easy to execute a Linux command and then use the resulting information within your program. For example, your program may need to know the hostname of the server that it's currently running on. Here is an example:

```
my $hostname = `hostname`;  
print "The current hostname is: $hostname\n";
```

Similarly, you can use any Linux command in a Perl program by putting the name of the command in single back quotes. These are not the same quotes that are on the key with double quotes; these quotes are found in the upper left corner of the keyboard above the Tab key.

Here is another interesting use of system interaction. Typing the `ENV` command at a Linux prompt will display a long list of all the system environmental variables. Also, `ENV` is an array that all programs have available to them. This includes the user, operating system, home directory, mail directory, and many other variables. If you had an executable of a Perl program you'd written that is only authorized to be run by certain individuals, you could test the `USER` variable held in the `ENV` array. For instance, here is how I could cause the program to terminate unless I am the one attempting to execute it:

```
if ($ENV{USER} eq "aebuell")  
{  
    print "You are an authorized user\n";  
}
```

```

else
{
    die ("You are not authorized to use this program");
}

```

Die is a Perl function that kills the program for you and can display a message if you choose. We will also see later how `die` see how it is also useful when working with files.

Input From User

If your program needs input from a user, this is pretty simple to accomplish in Perl.

```

print "Please enter a response: \n";
$response = <STDIN>;

```

The above line reads from standard input; i.e., the terminal. The program will stop until the user enters something at the prompt and hits enter. Then the response is held in the variable `$response` and can be used in the program.

Additionally, a Perl program can be run with command line arguments. This is done by typing the file name as you normally would to execute it, and then typing the arguments after the file name with a space between each, no commas (you should know in advance how many and what kind of arguments the program is expecting). Command line arguments, like C/C++, are held in an array called `ARGV`. Here is a short little program to illustrate its use:

```

#!/user/bin/perl

if ($ARGV[0])
{
    print "ARGV array = @ARGV\n";
}

else
{
    print "No command line arguments in ARGV\n";
    die("Program terminating. \n");
}

```

Copy and paste the above code into a text editor, save it as `example1.pl`, and then execute it with a number or string as an argument, for instance, like this:

```

perl example1.pl testing

```

The program should print out the word `testing`.

Working with Files

Here is how you would open a file called `datafile1` for reading:

```
open(INFILE, "<datafile1");
```

The name INFILE is a variable name of your choice. The contents of the file can be read into an array like this:

```
@array = <INFILE>;
```

After which you should close the file with the statement:

```
close(INFILE);
```

Here is how you would open a file called data1 for writing:

```
open(OUTFILE, ">data1");
```

Notice this is very similar to opening a file for reading, but the ">" sign is used instead of the "<" sign.

In the next section we'll see how to clean up data that is read in from files.

Data Manipulation

Besides being known for writing CGI scripts, Perl is also an excellent language for data manipulation. It's got some really interesting functions built in that are extremely useful when reading, processing, and writing data sets. For instance:

Chomp

The **chomp** function takes a string and removes all newline characters. This is most useful when you've read some data from a file into an array, and want to do some processing with that data. Almost always, the data will contain newlines, and almost always, you'll want to remove them before you work with the data.

datafile1 is a plain text file that contains 7 lines, each of which is the name of a division in a company. It looks like this:

```
Information Technology
Research and Development
Marketing
Accounting
Sales
Management and Supervision
Public Relations
```

Here is an example of reading the data into an array:

```
open(INFILE, "<datafile1") || die("cannot open datafile1");
@array = <INFILE>;
chomp(@array);           #removes the newlines
```

Now each element in the array contains the name of one of the divisions of the company and this data can be used more conveniently.

split

The `split` function takes two arguments, a regular expression between backslashes, and a string. It then breaks that string up into array elements based on the expression. Here is an example that uses a simple expression:

```
my $string = "1-2-3-4-5-6";
my @array = split(/-/, $string);
```

The second line takes the variable `$string` and splits it at each occurrence of the "-". Now, `@array` has six elements, 1 through 6.

join

The `join` function merges elements together. Here is an example:

```
my @letters = (a..k);    #the array contains letters a thru k
my $glue = join(' ', @letters);
print "$glue";
```

This code will print `a b c d f g h i j k`

Note that while the `split` function basically takes a scalar and makes it into an array, the `join` function takes an array and makes it into a scalar.

splice

This function inserts and/or deletes at selected locations in a string. It takes two arguments, an array and an offset. There are two more additional arguments, a length and a list. It looks like this:

```
splice(@array, offset, length, list);
```

The function removes elements from the array beginning at the offset, and a list of the removed array elements is returned. If `length` is specified, it only removes that many elements, otherwise, it removes them all until it reaches the end of the array. If `list` is specified, then the removed elements are replaced with the elements in the list.

Here is an example program that creates three new arrays by merging the array containing the contents of `rfile1` to the beginning of, the end of, and the (approximate) middle of the array containing the contents of `rfile2`.

You can test the program by creating the two text files, `rfile1` and `rfile2`, and placing them in the `cgi-bin` with your Perl programs.

`rfile1` contains the following lines of text:

```
1
2
3
4
5
6
```

`rfile2` contains the following lines of text:

a
b
c
d
e

```
#!/opt/local/bin/perl

use strict;
use warnings;

open(INHANDLE1, "rfile1") or die("cannot open rfile1\n");
open(INHANDLE2, "rfile2") or die("cannot open rfile2\n");

my @array1 = <INHANDLE1>;
my @array2 = <INHANDLE2>;

chomp(@array1, @array2);          # remove newlines and make
my @copy_array2 = @array2;       # copies so that the
my @second_copy_array2 = @array2; # contents of rfile2 are
my @third_copy_array2 = @array2; # not destroyed

splice(@copy_array2, 0, 0, @array1); # beginning
splice(@second_copy_array2, 2, 0, @array1); # middle
splice(@third_copy_array2, 5, 0, @array1); # end

print "@copy_array2\n @second_copy_array2\n"
```

The preceding code prints out:

```
1 2 3 4 5 6 a b c d e
a b 1 2 3 4 5 6 c d e
a b c d e 1 2 3 4 5 6
```

You can accomplish some pretty impressive and complicated-looking tasks with these functions, but before moving beyond the basics you should be familiar with regular expressions. This is an entire topic in and of itself, so I won't go into it here, but if you haven't used regular expressions before, here are a couple of references:

Regular Expressions Tutorial <http://www.grymoire.com/Unix/Regular.html>
Regular Expression Pocket Reference By Tony Stubblebine, published by O'Reilly

CGI

CGI stands for Common Gateway Interface and is a standard way for data to be passed between web applications, for instance, passing data from an online HTML form to a script on the server. CGI is not to be confused with a programming language.

It is a server-side communication standard supported by all web servers for accessing external programs. Examples of CGI programs are gateways to databases and scripts that process and return HTML commands to the server. Since HTML allows only one-way communication from the

server, which is read by the web browser or client, CGI permits communication and interaction from the client to the server for two-way, dynamic web pages.

Each time you view a web page, the client (your browser, like Internet Explorer or Firefox) requests information of the web server, which sends it the information and then closes the communication. The browser takes the received information and displays it as a web page that you can view.

When your browser requests a web page that uses a CGI program, the server calls the CGI program which generates a web page using the information it needs, and sends it back to your browser.

CGI programs can actually be written in any language, like C or Lisp. Perl is the most commonly used language for CGI however, because it is adept at dealing with text.

Forms

CGI is very commonly used with HTML forms on the web. Everyone who's browsed the Internet at some time or another has encountered a CGI form. They're used for guest books, surveys, shopping carts, comment forms, and much more.

To use forms on a web page, you need a little familiarity with HTML. I won't go into HTML here, but you should have a little familiarity with it in order to understand certain parts of the CGI scripts. Here is a good tutorial:

<http://www.htmlgoodies.com/primers/html>

I'll do my best to make sure that any HTML code I demonstrate here is well-formed and can be considered XHTML code. (XHTML is exactly like HTML code but it MUST be well-formed).

Example 1

Here is an example of a simple form written in XHTML:

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html >
4    <head>
5      <title>Form Example</title>
6      <link rel="stylesheet" type="text/css" href = "mai n. css" />
7    </head>
8    <body>
9      <form action = "http://mypage.iusb.edu/cgi-
10     bin/cgiwrap/aebuell/submi t. pl " method = "get">
11        Your Name <input type = "text" name = "name" /><br />
12        Comment<br />
13        <textarea name = "comment" rows = "5" cols = "40">
14        </textarea> <br />
15        <input type = "submi t" value = "Submi t" /><input type = "reset"/>
16      </form>
17    </body>
    </html >

```

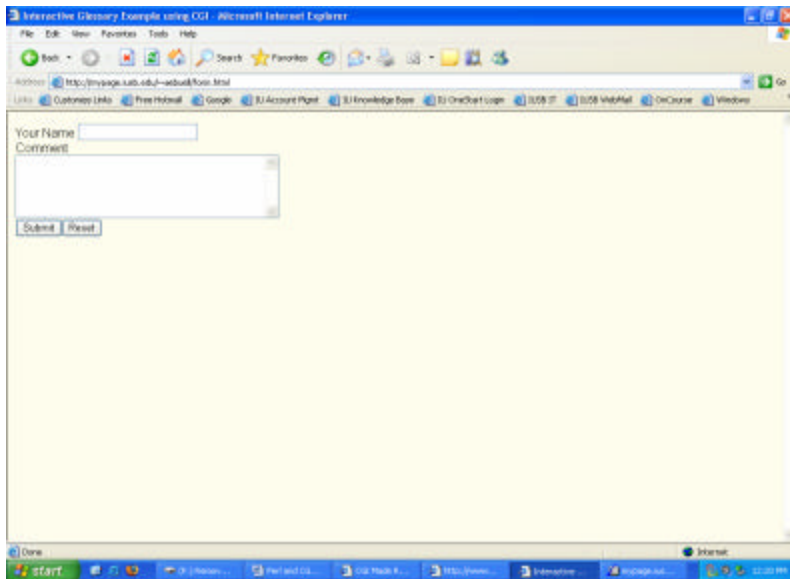
The code can be copied and pasted into a text editor and saved as form.html. This is part of a

web page I wrote and happens to link to an external style sheet file. You can either completely delete line 6 or copy the style sheet at <http://mypage.iusb.edu/~aebuell/main.css> and place it your html directory.

Also, line 9 needs to be changed to reflect your user name. This is where we have to go through the CGIWrap program in order to successfully call the CGI program. This is not necessary on the CS server or other servers. Detailed instructions are found at:

<http://mypage.iusb.edu/cgiwrap/setup.html>

Next set the permissions to be world readable and executable. You can then view form.html in a browser. My form is at <http://mypage.iusb.edu/~aebuell/form.html>. It looks like this:



The first eight lines of the code are not terribly important. If you wanted to keep it really simple, they could be condensed to:

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

The actual form, which is what we are concerned with for the moment, starts at line 9. When the form is submitted, the action attribute requests the server to execute the Perl file called submit.pl. The method attribute can be set to either `get` or `post`. The default is `get`. It really makes no difference for our purposes.

Lines 10 through 14 create the types of input seen in the form. Another example of an input type is a radio button. If you wanted to include this in a form, you would use the word "radio" as the input type.

When the form is submitted, the CGI program is contacted and the data that was entered into the form is passed to the program, in this case `submit.cgi`, which uses the data to do whatever it is that its supposed to do, and sends a response back to the browser, which in turn displays the web page accordingly.

Example 2

Here's another example of XHTML code for a guestbook:

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html >
4    <head>
5      <title>Form Example</title>
6      <link rel="stylesheet" type="text/css" href = "main.css" />
7    </head>
8    <body>
9
10   <form method="post" ACTION="http://mypage.iusb.edu/cgi-
11   bin/cgiwrap/aebuell/guestbook.pl ">
12
13     <input name="name" size=50 type="text"> <B>Your Name</B><BR>
14     <input name="email" size=50 type="text"> <B>Your E-Mail
15     Address</b><br />
16     <input type="hidden" name="submitaddress"
17     value="abuell@cs.iusb.edu">
18     <strong>Tell me your comments: </strong><p />
19     <textarea name="feedback" rows=10 cols=50></textarea><p />
20
21     <center>
22     <input type=submit VALUE="SEND">
23     <input type=reset VALUE="CLEAR">
24     </center>
25
26   </form>
27 </body>
28 </HTML>

```

The above form is using basically the same elements that we saw in the first example, with a few additional attributes such as size, etc, to provide a little more formatting. The Perl script called by the HTML form is on the following page.

```

1  #!/usr/bin/perl
2
3  if ( $ENV{'REQUEST_METHOD'} eq 'POST' )
4  {
5      read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
6      @pairs = split(/&/, $buffer);
7
8      foreach $pair (@pairs)
9      {
10         ($name, $value) = split(/=/, $pair);
11         $value =~ tr/+ / /;
12         $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
13
14         $FORM{$name} = $value;
15     }
16
17 # ***** Send E-mail *****
18
19     open (MESSAGE, "| /usr/lib/sendmail -t");
20
21     print MESSAGE "To: $FORM{submitaddress}\n";
22     print MESSAGE "From: $FORM{name}\n";
23     print MESSAGE "Reply-To: $FORM{email}\n";
24
25     print MESSAGE "Subject: Feedback from $FORM{name} at
26 $ENV{'REMOTE_HOST'}\n\n\
27 n";
28     print MESSAGE "The user wrote: \n\n";
29     print MESSAGE "$FORM{feedback}\n";
30     close (MESSAGE);
31
32     &thank_you;
33 }
34
35 # ***** Creates Dynamic Output *****
36
37 sub thank_you
38 {
39     print "Content-type: text/html\n\n";
40     print "<html >\n";
41     print "<head>\n";
42     print "<title>Thank You page</title>\n";
43     print "</head>\n";
44     print "<h1>Thanks for your feedback. h1>\n";
45     print "\n";
46     print "<p>\n";
47     print "<h3>Your comments will be reviewed.<br />\n";
48     print "<p />\n";

```

```

49     print "</body>\n";
50     print "</html >\n";
51     exit(0);
52 }

```

You can see the script and test it out here:

<http://mypage.iusb.edu/~aebuell/guestbook.html>

The code may look complicated but most of this script consists of things we've seen before.

3: When an HTML form calls a CGI program, the data is passed in a long string that looks like:

```
name1=value1&name2=value2&name3=value3
```

To access this string, it depends on whether the method used in the form action was post or get. In our example, we used post. That means that the long string of values is accessible in the environmental variable `CONTENT_LENGTH`. If `get` is used, then the data is in an environmental variable called `QUERY_STRING`.

5: `read` is a Perl function. It has form

```
read(FILEHANDLE,SCALAR,LENGTH)
```

It reads `LENGTH` bytes of data into variable `SCALAR` from the specified `FILEHANDLE`. It returns the number of bytes read, 0 at end of file, or `undef` if there was an error.

If you'll remember, `@ENV` is an array, and when an array is evaluated in scalar form, it returns the length of the array. This is the purpose of the term

```
$ENV{'CONTENT_LENGTH'}
```

So, the statement

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'}
```

is reading the length of `CONTENT_LENGTH` from `STDIN` into `$buffer`.

6: Now `$buffer` holds the long string that contains the data values, which as mentioned in the explanation of line 3, looks like this:

```
name1=value1&name2=value2&name3=value3
```

Line 6 splits this string on the "&" sign and assigns it to `@pairs`.

8-15: This function further splits the string on the "=" into names and values. Lines 11 and 12 convert all "+" characters to spaces, and convert all "%xx" sequences to the single character whose ascii value is "xx", in hex.

Then, the names and values are placed in a hash called `FORM`

19: Opens a file handle called `MESSAGE` that is a pipe to the `sendmail` program that's on the server. The "|" sign in the code below stands for pipe.

```
" | /usr/lib/sendmail -t"
```

21-30: Prints to the filehandle MESSAGE the contents of the data from the form, and then closes the filehandle.

32: Calls the function that creates the HTML for the thank-you page.

39-51: These lines are simply using the print function to print HTML code to the client. The program then exits with a code of 0.

Hopefully this gives you some understanding of the capabilities of Perl and CGI. I've touched on some more advanced topics but won't go into deeper detail here. Check out the references for further reading.

References

Bengry, S and Heller, E. XML: An Introduction. 3rd ed. Element K Content 2005.

Castro, Elizabeth. HTML For the World Wide Web. 5th ed. Peachpit Press 2003.

Christenberr, J. et al. CGI Fast & Easy Web Development. Prima Publishing 2003.

Comprehensive Perl Archive Network <http://www.cpan.org>

D Foy, Phoenix, and Shwartz. Learning Perl. 4th ed. O'Reilly 2005.

O'Reilly's Perl.com <http://www.perl.com>

Pierce, Clinton. Teach Yourself Perl in 24 Hours. 3rd ed. Sams Publishing 2005.

Appendix A

Terms

ADS

Active Directory Service.

CGI

CGI stands for Common Gateway Interface. CGI allows HTML pages to interact with programming applications.

HTML

Hypertext Markup Language. The coded format language used for creating hypertext documents on the World Wide Web and controlling how Web pages appear.

Linux

Linux is a free open-source operating system based on Unix, originally created by Linus Torvalds. Linux has a reputation as an efficient and stable system.

Windows Server 2003

The successor to Windows 2000 Server, Microsoft's Windows Server 2003 (codename Whistler Server, also known as Windows NT 5.2) is one of Microsoft's server operating systems, the latest version as of 2006.

Unix

Unix is an operating system co-created by AT&T researchers Dennis Ritchie and Ken Thompson. Unix is well known for its relative hardware independence and portable application interfaces.

XML

Short for Extensible Markup Language, a specification developed by the W3C. XML is a version of SGML, as is HTML. XML and HTML can be thought of as siblings, with SGML as the parent. XML is called extensible because users can create their own tags.

Appendix B

Trouble-Shooting Resources

Syntax Errors

There are the simplest to fix. It's common to forget to close a brace, include a comma, or use a comma where you should've used a semi-colon. The Perl error message may or may not be indicative of this kind of error, but at least it'll point you to the right line, which you can examine closely.

Make sure to include the statements

```
use warnings;  
use strict;
```

In the beginning of your program; at times this can be immensely helpful.

Print, Print, Print

If you're certain you don't have a syntax error, you must have an error in logic, which can be more difficult to debug. An elementary but effective way to debug programs is to print stuff out in every place that looks suspect. Somewhere, your variables don't contain what you expect them to. Alternatively, you can use the debugger. Sometimes it is more convenient and sometimes not. I personally rarely use it.

The Debugger

The Perl interpreter includes a debugger that can be useful while developing a Perl program. It allows you to step through a program line by line so you can pinpoint the spot where something goes wrong.

To start the debugger, use the `-d` option when executing a Perl program, like so:

```
perl -d program-name
```

Now you will see some information displayed such as the version number, and you will have a prompt. Type `help` to get a list of all available debugger commands. Use the `n` command to execute one instruction at a time. Continue to use `n` to step through the program. At any point, you can type `print @array` to check what variables are in an array. To insert a breakpoint, type `b` and the line number where you want the debugger to stop program execution, like this:

```
b 33
```

As long as line 33 is valid code, the debugger will pause the execution at that line. You can't insert a breakpoint at a brace or blank line. Now, if you want to continue on, just type `c`.

This is just a brief introduction. For more information on the debugger, visit the GNU Data Debugger web page:

<http://www.gnu.org/software/ddd/>

Problems Using CGI

If you get an error that CGI.pm or any other module can't be located, make sure you've typed the module name correctly. If that's not the problem, your installation may be incomplete or corrupted. This can probably only be fixed by a system administrator.

If you get an Error 404 File Not Found error, you probably typed an URL wrong or placed the CGI script in the wrong directory.

If you get an error that says Forbidden, you haven't set your file permissions correctly. Make sure that the file in question has execute and read permission for everyone.

Research

By typing perldoc perl at a command prompt, you can see all available documentation, including Perl FAQ's, which is worth looking over. You might find your problem. Keep a Perl text on hand for reference, or do a google search for information.

Ask For Help

Lots of people post technical or programming questions on message boards or newsgroup discussions and hope that someone will be able to help them out. Here are some Perl newsgroups accessible from groups.google.com:

comp.lang.perl.moderated
comp.lang.per.misc

Make sure to read the group's FAQ before posting a message. Also check out these message boards:

<http://www.webdeveloper.com/forum/>