Open Source Software Project Development

Dr. T.Y. Wong

Weeks 2 & 3

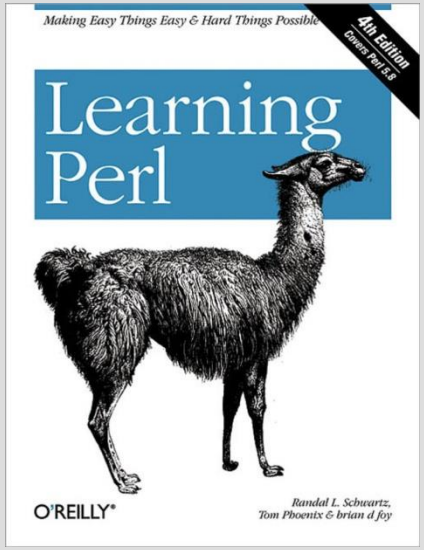
Scripting VS Interpreted Languages & Perl.

- The good, the bad, and the ugly of Perl...

References.

- Learning Perl
  - Author: Randal L. Schwartz, et.al.
  - Publisher: O'Reilly.

- http://www.cpan.org/



Server-Side Programming

- to continue what we've started before...

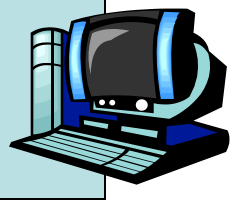
Two-side of Web Programming...

How to create request?
- reload?
- hyperlink?
- submit button in forms?
- asynchronous requests?

How to manage display?
- State-of-the-art:
 JavaScript + DOM manipulation.

How to manage client data?
- Login sessions?
- Cookies?

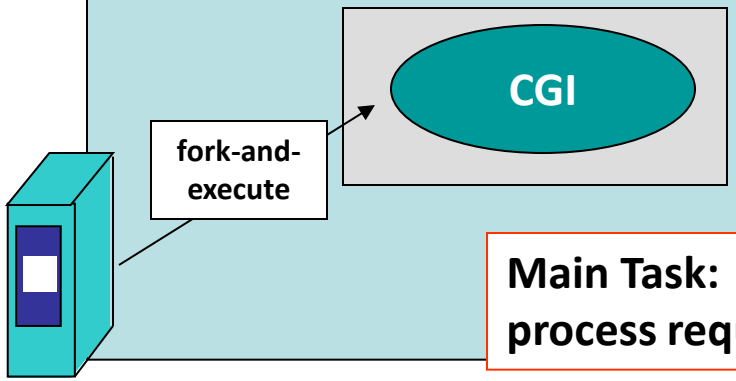
Main Task:
create request & manage display.



How to program?
- all languages can do so.
- mainstream: Perl, Python, Ruby, PHP, ASP, JSP.

Where are the data stored?
- text files?
- database?

Should the environment be bound?



Main Task:
process request!

Two-side of Web Programming...

Our focus:

Web server-side programming
!=
Scripting language

But, there are lots of outstanding features in scripting languages supporting web server-side programming.

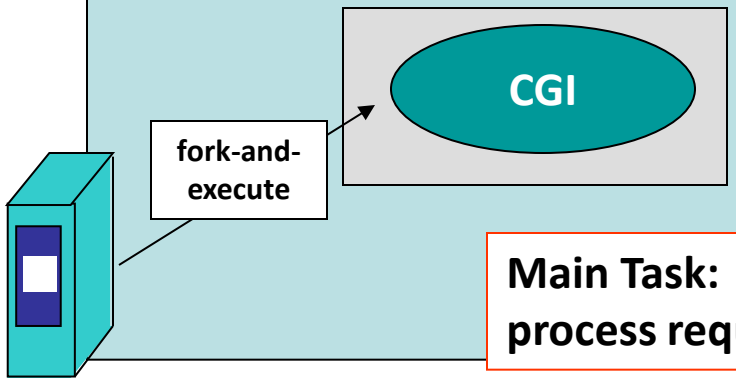
E.g., tons of modules in PHP and Perl;
The rail development pattern for Ruby;
etc.

Why not learning those new stuffs!

How to program?
- all languages can do so.
- mainstream: Perl, Python, Ruby, PHP, ASP, JSP.

Where are the data stored?
- text files?
- database?

Should the environment be bound?



Main Task: process request!

Program codes for basics

all_files.zip	array_copy.cgi	array_push_pop.cgi	array_shift_unshift.cgi
array_slice.cgi	array_sort_reverse.cgi	hash_example.cgi	perl_cat.cgi
perl_hello_world.cgi	perl_pipe.cgi	scalar_example.cgi	shell_script.cgi
string_example.cgi	---	---	---

Fall 2011, CSCI4140, Department of Computer Science and Engineering, The Chinese University of Hong Kong.

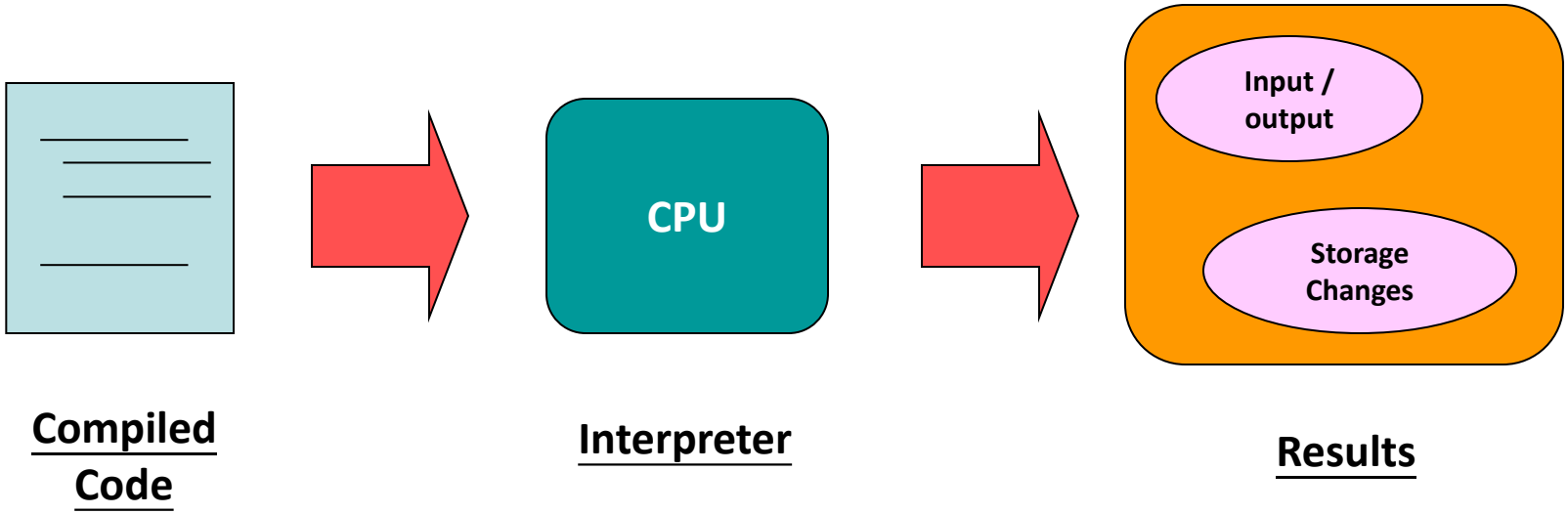
http://appsrv.cse.cuhk.edu.hk/~csci4140/cgi-bin/perl/script/basics/

Starting From the Basics

-The Scripting Languages...Shell Scripting

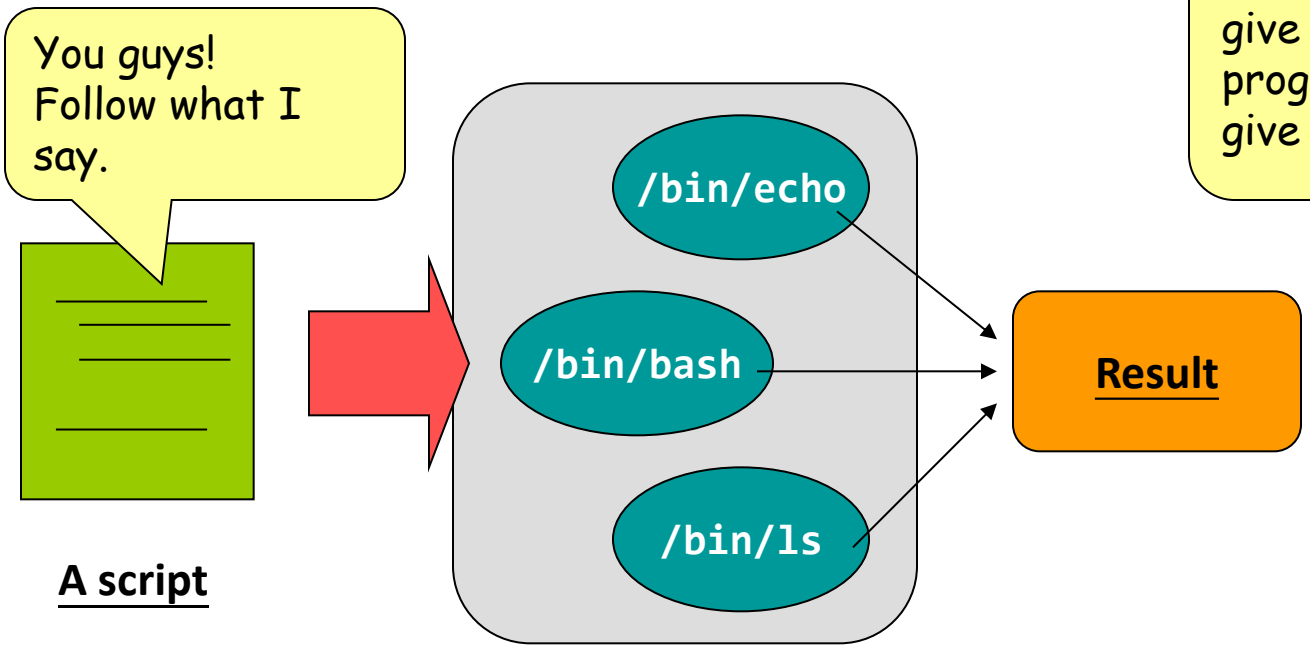
Compiled Language

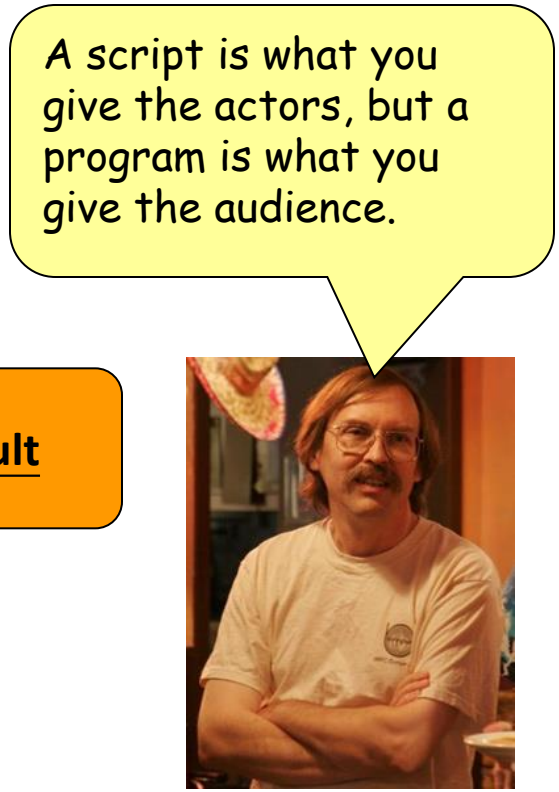
- In this university, we start our programming training with **compiled languages**: C, C++, Java, etc…



Scripting Language

- So, what is a scripting language?



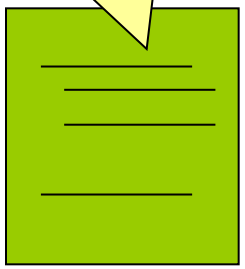


Larry Wall
Author of the programming language, Perl

Scripting Language

- So, what is a scripting language?

You guys! Follow what I say.



A script

So, a script is a sequence of orders.

Each order will be interpreted by the involved program individually.

Of course, **a script requires a interpreter** in order to dispatch the commands inside the script.

A script is what you give the actors, but a program is what you give the audience.

Larry Wall
Author of the programming language, Perl

Scripting Language

Step (1)

Execute the interpreter.
In this time, the interpreter is "**/bin/bash**".
"**/bin/bash**" will then be invoked.

```
#!/bin/bash

echo "hello world!"
date
```

Program: script.sh

The interpreter then reads the script file **line-by-line**.

Notes: comments starts with '**#**', except the first line.

Step (2)

/bin/bash

Step (3) and so on…

invoke: **echo** **"hello world"**

program name

argument

[Example] shell_script.cgi

Scripting Language

From the OS' point of view… and a simple view…

Program: script.sh

```
#!/bin/bash

echo "hello world!"
date
```

invoking: ./script.sh

equivalent

invoking: /bin/bash ./script.sh

equivalent

Start a new shell /bin/bash

executing the content of "./script.sh" in a line-by-line manner.

```
$ /bin/bash ./script.sh
or
$ ./script.sh
```

So, both have the same effect.

Shell scripting language highlights

- Syntax.
  - For each shell, it has its own set of syntax. E.g., "**`/bin/tcsh`**" and "**`/bin/bash`**" have different set of syntax.
  - One thing in common is that:

"**$name**" means reading the value of a variable "**name**".

E.g.

```
name="xxx";
namespace=$name
name=${namespace}
```

A semicolon means the end of a command. But, usually, there is no semicolon at the end. Instead, a newline defines the end of a command.

Curly braces are there to define the start and the end of the name.

Shell scripting language highlights

- Data type
  - Very funny, **only one variable type**:
  - **STRING!**

Interesting Fact

When you want to add two numbers, the following won't work!

"2" + "4"

A string adds to another string?!
Sorry, we are not writing in JAVA/C++...

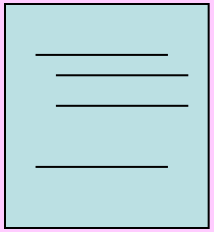
You need **an external program** to do the addition, e.g., using "**expr**".

result=`expr 2 + 4`

`.....` is called **sub-shell**. It means calling a shell to execute the command specified. Then, return the resulting string.

Compiled language VS Scripting…

Compiled code

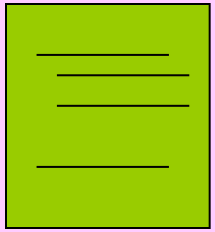


Language: very solid and rigid - the language itself is very restrictive. The language cannot be modified easily.

CPU

Interpreter: Powerful- as long as the CPU has the right to do the tasks.

Script



Language: very loose - the functionality can be easily extended by adding new commands.

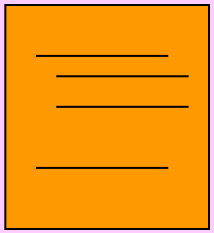
Shell

Interpreter: Restrictive - depend on the availability of the programs desired.

In between them, it is the interpreter language.

Interpreted language?

Script written in Interpreted Language



Language's relaxation - it is not a compiled language; it needs an interpreter.

The language is not as strong as that of a compiled language, but it is better than the scripting language.

Interpreter

Interpreter's ability - the interpreter is stronger than that in the scripting language.

It contains a lot of functions, including programming libraries.

Interpreter Language

-Perl, the world of Web applications starts here.

Perl – an interpreted language

- Some highlights:
  - Perl may stands for:
    - "Practical Extraction and Report Language", or
    - "Pathologically Eclectic Rubbish Lister".

Three basic data types: - scalar, array, and hash.	**An interpreter language.** No compilation is required.	**Dynamic typing.** A scalar variable does not have a type or it can be any types.
Multi-paradigm. It can be an imperative language or an OOP language.	**Structured programming language.** Support functions and libraries.	

Perl 101 – basic syntax

- A "quite" loose syntax:
  - The following shows **4 possible ways** to write a "hello world" program…

```
#! /usr/bin/perl -w

print "hello world\n";
print ("hello world\n");
print STDOUT "hello world\n";
printf ("hello world\n");
```

Tell the interpreter to show any warnings encountered.

Similar to most languages (but this is not a must in shell scripting)

"There's more than one way to do it!" Perl motto!

[Example] perl_hello_world.cgi

Perl 101 – basic syntax

- A "quite" loose syntax:
  - The following shows **4 possible ways** to write a "hello world" program…

```
#! /usr/bin/perl -w

print "hello world\n";
print ("hello world\n");
print STDOUT "hello world\n";
printf ("hello world\n");
```

Parentheses are **optional** for function calls.

[Example] perl_hello_world.cgi

Perl 101 – scalar

- A set of rich and useful data types.
  - scalar, array, and hash.

A name starting with '**$**' is a scalar: **number & string**.

```
#!/usr/bin/perl -w

$var1 = "123";          # a string
$var2 = 123;            # a number
$var3 = "${var1}456";   # variable inside a string.
$var4 = '${var1}456';

print "var1 = $var1\n";
print "var2 = $var2\n";
print "var3 = $var3\n";
print "var4 = $var4\n";
```

'$' means a scalar variable.

Dynamic typing: a scalar variable can be a string or a number…and, of course, the following statement is absolutely correct.

```
$var1 = $var2;
```

We won't find that in C because C adopts **strong typing**.

[Example] scalar_example.cgi

Perl 101 – scalar

- A set of rich and useful data types.
  - scalar, array, and hash.

Perl is tailor-made for operating over strings. **Double quotes** and **single quotes** are therefore have important roles to play…

```
#!/usr/bin/perl -w

$var1 = "123";          # a string
$var2 = 123;            # a number
$var3 = "${var1}456";   # variable inside a string.
$var4 = '${var1}456';

print "var1 = $var1\n";
print "var2 = $var2\n";
print "var3 = $var3\n";
print "v
```

When double quotes are used for bounding a string, **values of variables will be read** and be used inside a string.

When single quotes are used for indicating a string, the content will be displayed in the **verbatim mode**.

[Example] scalar_example.cgi

Perl 101 – scalar

- A set of rich and useful data types.
  - scalar, array, and hash.

```
#!/usr/bin/perl -w

$var1 = "123";          # a string
$var2 = 123;            # a number
$var3 = "${var1}456";   # variable inside a string.
$var4 = '${var1}456';

print "var1 = $var1\n";
print "var2 = $var2\n";
print "var3 = $var3\n";
print "var4 = $var4\n";
```

So, what will the output be?

```
var1 = 123
var2 = 123
var3 = 123456
var4 = ${var1}456
```

[Example] scalar_example.cgi

Perl 101 – scalar

- Perl is born to serve strings…

```
char *str = malloc(...);
strcpy(str, “hello ”);
strcat(str, “world”);
```

```
$str = “hello ” . “world”;
```

```
char *str1 = malloc(...);
char *str2 = malloc(...);
strcpy(str1, “hello”);
strcpy(str2, str1);
```

```
$str1 = “hello”;
$str2 = $str1;
```

```
if(strcmp(str1,str2)==0) {
    ......
}
```

```
if($str1 eq $str2) {
    ......
}
```

```
char *str = ......
strcpy(str, “hello”);
strcat(str, “hello”);
```

```
$str = “hello” x 2
```

C realm

Perl realm

[Example] string_example.cgi

Perl 101 – array

- A set of rich and useful data types.
  - scalar, array, and hash.

```
#!/usr/bin/perl -w

@array1 = ("hello", "world");
@array2 = qw(hello world);
```

They are the same!

"**qw**" stands for "*quote word*".

It automatically adds quotes to words, and inserts the words into the target array.

"Perl is difficult to learn," someone may say…

Encore!

"There's more than one way to do it!" Perl motto!

Perl 101 – array operations

- A set of rich and useful array operations…
  - copying…

```
@array = ("hello", "world");
@array = ("say", @array);
```

How about this?

```
#!/usr/bin/perl -w

@array1 = ("hello", "world");   # initialization
@array2 = @array1;              # copying
($str1, $str2) = @array2;       # copying again?!

print $str1, "\n";
print $str2, "\n";
```

This is an array declaration + initialization.

[Example] array_copy.cgi

Perl 101 – array operations

- A set of rich and useful array operations…
  - this following operation is called **slice**…
    - The elements in the new array is a sub-set of the elements of the old array.

```
@array = ("say", "hello", "world", "together");

print '@array:          ',   "@array\n";
print '@array[0,1]:     ',   "@array[0,1]\n";
print '@array[0,2]:     ',   "@array[0,2]\n";
print '@array[1,2]:     ',   "@array[1,2]\n";
print '@array[3,2,1,0]: ',   "@array[3,2,1,0]\n";
print '@array[0..1]:    ',   "@array[0..1]\n";
print '@array[0..2]:    ',   "@array[0..2]\n";
print '@array[1..3]:    ',   "@array[1..3]\n";
```

[Example] array_slice.cgi

Perl 101 – array operations

- Array: **push()** & **pop()**.

Treat an array as a stack, then…

The top-most element of the stack has the largest index.

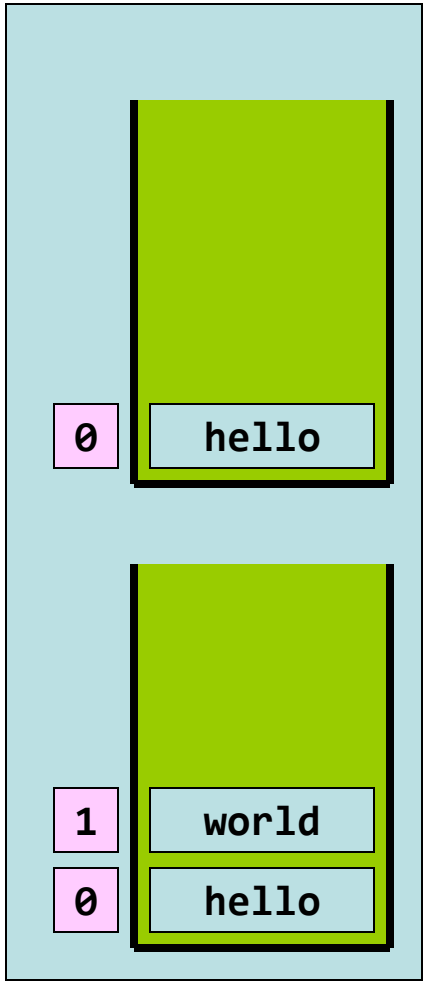
```
@array = ();
push(@array, “hello”);
push(@array, “world”);

$len = scalar @array;
for($i = 0; $i < $len; $i++) {
    print $array[$i], “\n”;
}
print “\n”;

while ( $i = pop(@array) ) {
    print $i, “\n”;
}
```

To read the length of an array. Weird…

An array element is a scalar, and ‘$’ should be used.



[Example] array_push_pop.cgi

Perl 101 – array operations

- Array: **`shift()`** & **`unshift()`**.

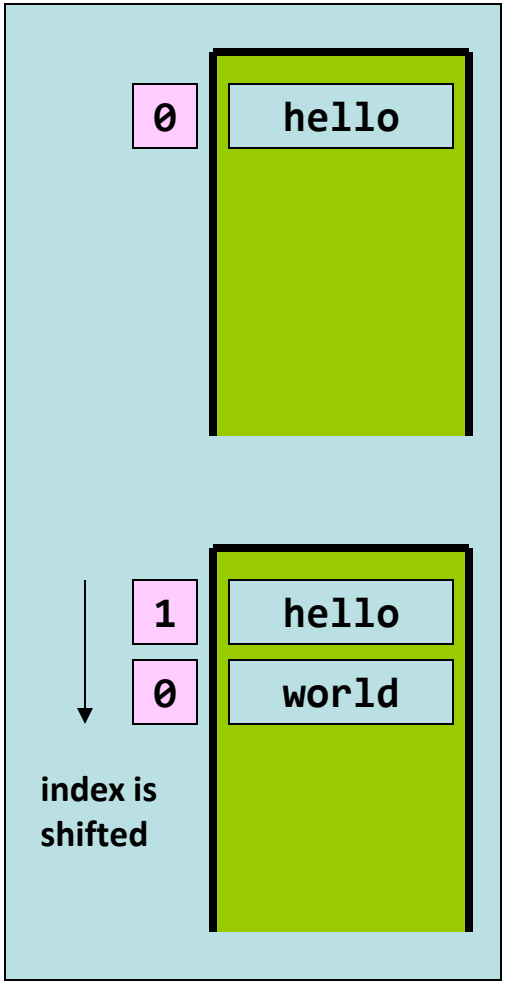
This time…the stack operations is reversed….

The top-most element of the stack has the index 0.

```
@array = ();
unshift (@array, “hello”);
unshift (@array, “world”);

$len = scalar @array;
for($i = 0; $i < $len; $i++) {
    print $array[$i], “\n”;
}
print “\n”;

while ( $i = shift(@array) ) {
    print $i, “\n”;
}
```



[Example] array_shift_unshift.cgi

Perl 101 – array operations

- Array: **`sort()`** & **`reverse()`**.

```
@array = ("say", "hello", "world", "together");

@new_array = sort(@array);
print "sorted:\n";
foreach $content (@new_array) {
        print "$content ";
}
print "\n\n";

@new_array = reverse(@array);
print "reverse, sorted:\n";
foreach $content (@new_array) {
```

"**foreach**" is an extremely useful for-loop: you don't need the array index.

It starts being implemented in shell script and is implemented in many languages now.

[Example] array_sort_reverse.cgi

Perl 101 – array operations

- A challenge!
  - How to implement a queue using array with...
    - **`shift() & unshift()`**.
    - **`push() & pop()`.**

Perl 101 – hash

- A set of rich and useful data types.
  - scalar, array, and hash.

An array using a scalar as an index.

```
$hash{"a"} = "say";
$hash{"b"} = "hello";
$hash{"c"} = "world";
$hash{"d"} = "together";

@temp = keys(%hash);
print "keys: @temp\n";

@temp = values(%hash);
print "values: @temp\n";

foreach $i ( keys(%hash) ) {
    print "$i $hash{$i}\n";
}
```

They are called keys, instead of indices.

Return all the keys as an array.

Return all the stored values as an array.

WOW!

[Example] hash_example.cgi

Perl 101 – hash

- A set of rich and useful data types.
  - scalar, array, and hash.

An array using a scalar as an index.

```
$hash{"a"} = "say";
$hash{"b"} = "hello";
$hash{"c"} = "world";
$hash{"d"} = "together";

@temp = keys(%hash);
print "keys: @temp\n";

@temp = values(%hash);
print "values: @temp\n";

foreach $i ( keys(%hash) ) {
    print "$i $hash{$i}\n";
}
```

	Array	Hash
Name starts with...	**@**	**%**
Index type	Integers >= 0	Scalar
Element type	Scalar	Scalar
Accessing elements	**$name[index]**	**$name{index}**

[Example] hash_example.cgi

Perl 101 – control structure

- A rich set of control structures.

```
if ( cond ) { ... }
if ( cond ) { ... } else { ... }
if ( cond ) { ... } elsif { ... }
```

Guess what? Unlike C, **you can't omit the pair of curly brackets** even though the body of the if-clause only contains one statement!

```
if ( 1 )
    print "true";
```

Syntax error!

```
if ( 1 ) {
    print "true";
}
```

Syntax OK!

Perl 101 – control structure

- A rich set of control structures.

```
if ( cond ) { ... }
if ( cond ) { ... } else { ... }
if ( cond ) { ... } elsif { ... }
```

```
while ( cond ) { ... }
until ( cond ) { ... }
for ( initial ; testing ; iteration ) { ... }
foreach var1 ( var2 ) { ... }
```

Well..they are opposite…

Perl 101 – relational operators

- For scalars only:

String comparison		Numeric comparison	
`ne`	`eq`	`!=`	`==`
`lt`	`gt`	`<`	`>`
`le`	`ge`	`<=`	`>=`

Very similar to C, there is no boolean type.

0 – for false

1 – for true

Btw, "**`undef`**" stands for undefined.

Perl 101 – File I/O

- File I/O in Perl is **AMAZING**…

```
#!/usr/bin/perl -w

while( $input = <STDIN> ) {
    print STDOUT $input;
}
```

"**<XXX>**" means **reading from the opened stream "XXX"**.

The print subroutine can **print to any opened streams**.

What? This is the program "cat"? So compact!

[Example] perl_cat.cgi

Perl 101 – File I/O

Opening regular files...

`open(STREAM_NAME, “xxx”);`	→	`STREAM_NAME = fopen(“xxx”, “r”);`
`open(STREAM_NAME, “> xxx”);`	→	`STREAM_NAME = fopen(“xxx”, “w”);`
`open(STREAM_NAME, “>> xxx”);`	→	`STREAM_NAME = fopen(“xxx”, “a”);`

Opening pipe files...

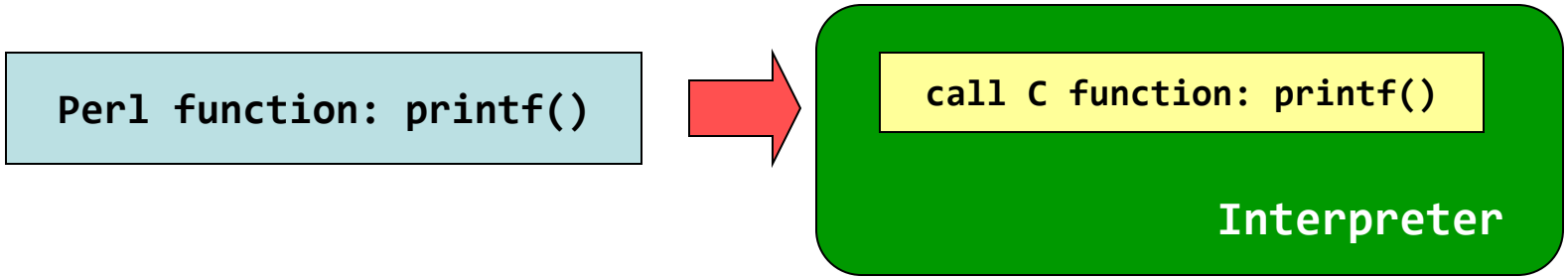
`open(STREAM_NAME, “| cmd”);`	→	`STREAM_NAME = popen(“xxx”, “w”);`
`open(STREAM_NAME, “cmd |”);`	→	`STREAM_NAME = popen(“xxx”, “r”);`

[Example] perl_pipe.cgi

More Perl?

- Wiki, Google, manpage, etc…

- You may notice that **Perl and C** have so many similarities…why is that?
  - Because Perl functions **wraps** C functions!
  - We call those Perl functions: wrappers…



Program codes for regex

all_files.zip	regex_file_ext.cgi	regex_match_ip.cgi	regex_rename_ext.cgi
regex_simple.cgi	---	---	---

Fall 2011, CSCI4140, Department of Computer Science and Engineering, The Chinese University of Hong Kong.

http://appsrv.cse.cuhk.edu.hk/~csci4140/cgi-bin/perl/script/regex/

Perl's True Color

- Regular Expression and pattern matching.

Perl's True Power

- In the world of string processing, **Perl is the king**!
  - Because it has a complete support for regular expression operations.
  - Even PHP and JavaScript have Perl-style regular expression support!

- Regular expression is a way for you to do
  - pattern matching;
  - pattern extraction;
  - pattern replacement.

Pattern Matching

- For example, we want to detect if a string contains the phrase "***OK***".
  - In C, you need to do tons of work:
    - **`malloc()`**, careful bound-checking, etc…
  - But, in Perl…

"**/OK/**" specifies the pattern that you want to match, i.e., OK.

```
#!/usr/bin/perl -w

$_ = <STDIN>;
if(/OK/) {
        printf("Matched\n");
}
else {
        printf("Not Matched\n");
}
```

"**$_**" is known as the "*default input and pattern-searching place holder*"

[Example] regex_simple.cgi

Pattern Matching

- More cases (1):
  - Suppose an input is a filename.
  - We want to match if the extension is "**txt**".

```
#!/usr/bin/perl -w

$_ = <STDIN>;
if(/txt/) {
        printf("Matched\n");
}
else {
        printf("Not Matched\n");
}
```

WRONG

We need extra knowledge.

It can match:

hellotxt

txt

hello.txt.jpg

Pattern Matching

- Special matching patterns (1 of 3):

.	Represent one character. So, "`/..`/" matches any strings with length >= 2.
x*	Match '**x**' for "*greater than or zero*" times. So, "`/x*/`" matches any strings.
x?	Match '**x**' for "*zero or one*" time. So, "`/ab?c/`" matches "ac", "abc", but not "abbc".
x+	Match '**x**' for "*at least once*". So, "`/x+/`" matches strings containing at least one '**x**' character.
^x	Match '**x**' at the start of a string.
x$	Match '**x**' at the end of a string.

Pattern Matching

- Special matching patterns (2 of 3):

[xyz]	Represent one character which must be either x, y, or z. i.e., "`[]`" includes a set of matching characters.
[^xyz]	Represent one character which must not be neither x, y, nor z. i.e., "`[^]`" includes a set of characters to be excluded.
[a-z]	Match a character from 'a' to 'z'. Similar patterns: `[A-Z]` and `[0-9]`.
x{5}	Match '`x`' for exactly 5 times.
x{5,}	Match '`x`' for at least 5 times.
x{0,5}	Match '`x`' for 0-5 times.

Pattern Matching

- Special matching patterns (3 of 3):

\.	Represent the character '.'
\d	Represent one digit. "**/\d/**" is the same as "**/[0-9]/**".
\w	Represent one alphabet. "**/\w/**" is the same as "**/[a-zA-Z]/**".
\s	Represent a space, a tab character, or a newline character. "**/\s/**" is the same as "**/[\r\t\n]/**".
\D	Opposite to "**\d**".
\W	Opposite to "**\w**".
\S	Opposite to "**\s**".

Pattern Matching

- More cases (1):
  - Suppose an input is a filename.
  - We want to match if the extension is "**txt**".

```
#!/usr/bin/perl -w

$_ = <STDIN>;
if(   ???   ) {
        printf("Matched\n");
}
else {
        printf("Not Matched\n");
}
```

The answer:

`/.*\.txt$/`

`/.+\.txt$/`

If "**.txt**" is not considered as a normal filename.

[Example] regex_file_ext.cgi

Pattern Matching

- More cases (2):
  - Test if the input is an IP address.

Answer:

```
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/
```

Note very important that:

We will accept wrong IP addresses such as “1.2.3.444”.
We need to extract the matched pattern!

Pattern Extraction

- In Perl, matched patterned patterns can be extracted using parentheses and be stored in an array.

```
$_ = <STDIN>;
@array = /^(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})$/;
if(@array) {
    foreach $i (@array) {
        if($i > 255) {
            printf("Pattern matched, but wrong value\n");
            exit 1;
        }
    }
    printf("Matched\n");
}
else {
    printf("Pattern not match\n");
}
```

[Example] regex_match_ip.cgi

Pattern Substitution

- For example,
  - we want to change the filename with ".mp3" extension to ".jpg" extensions…
  - in order that no one will discover I stored mp3 file in our personal account.
  - How to get it done in Perl?

Method:

```
s/pattern 1/pattern 2/
```

The statement substitutes "**pattern 1**" with "**pattern 2**".

The result will be stored in "$_".

Pattern Substitution

- For example,
  - we want to change the filename with ".mp3" extension to ".jpg" extensions…
  - in order that no one will discover I stored mp3 file in our personal account.
  - How to get it done in Perl?

Answer?

```
s/mp3/jpg/
```

You'll change "**mp3.mp3**" into "**jpg.jpg**", instead of "**mp3.jpg**".

Pattern Substitution

- For example,
  - we want to change the filename with ".mp3" extension to ".jpg" extensions…
  - in order that no one will discover I stored mp3 file in our personal account.
  - How to get it done in Perl?

Answer?

```
s/(.+)\.mp3$/$1\.jpg/;
```

Pattern matched inside the parentheses will be copied to the place that holds "**$1**". The second matched pattern will be "**$2**", and so and so for.

[Example] regex_rename_ext.cgi

Program codes for CGI

all_files.zip	join.cgi	parseCGI_step1.cgi	parseCGI_step2.cgi
parseCGI_step3.cgi	perl_tr_example.cgi	read_env_ver1.cgi	read_env_ver2.cgi
split.cgi	---	---	---

Fall 2011, CSCI4140, Department of Computer Science and Engineering, The Chinese University of Hong Kong.

http://appsrv.cse.cuhk.edu.hk/~csci4140/cgi-bin/perl/script/CGI/

Perl and CGI

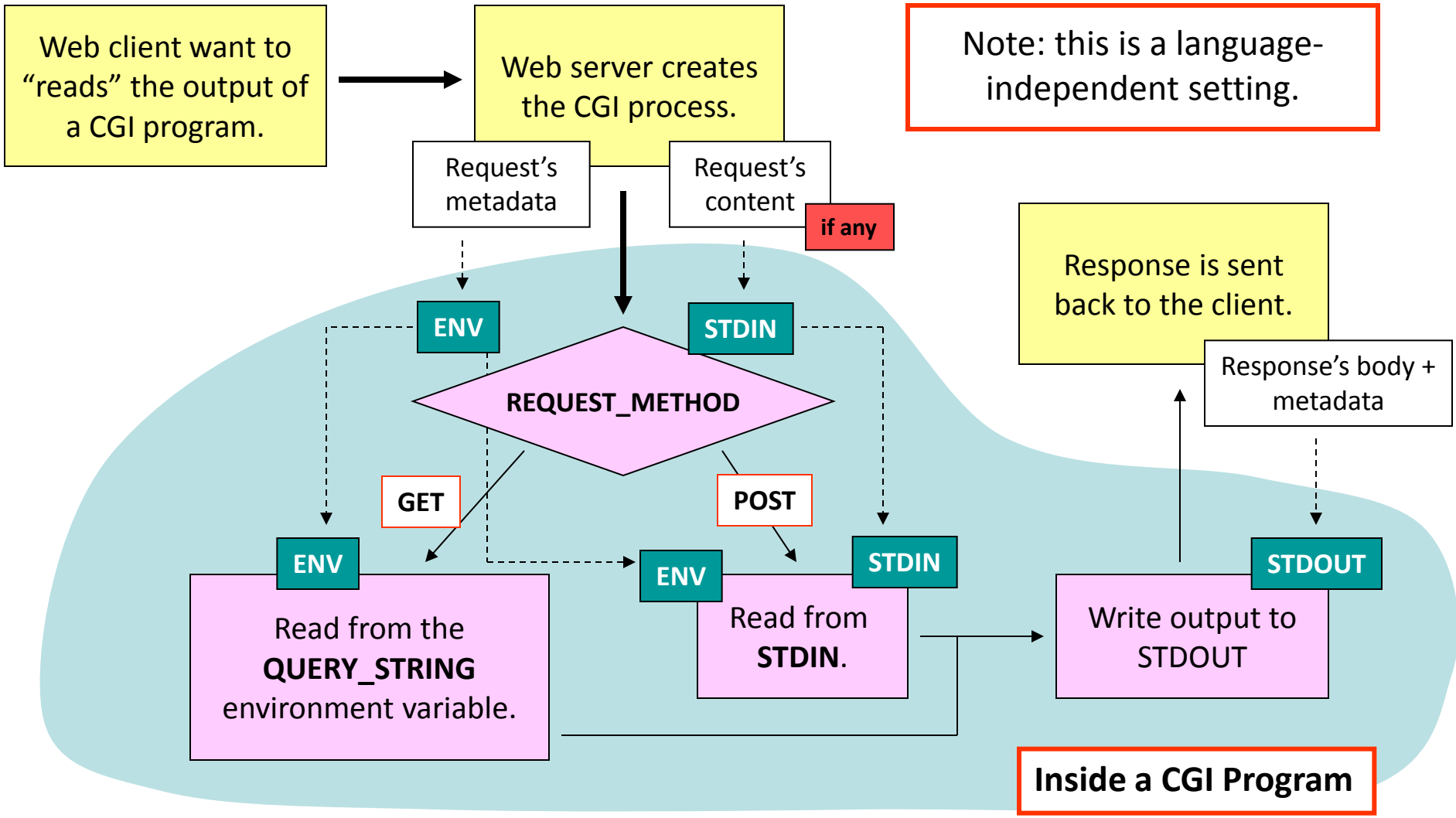
- How fit are they? Parsing & Validating inputs.

Perl and CGI

- How fit are they?
  - A perfect couple!

- Outstanding issues on CGI:
  - How to read the user input under different methods in Perl?
  - How to parse the user input in Perl?

Reminder…


Web client want to "reads" the output of a CGI program.
Web server creates the CGI process.
Note: this is a language-independent setting.
Request's metadata
Request's content
if any
ENV
STDIN
REQUEST_METHOD
GET
POST
ENV
ENV
STDIN
STDOUT
Read from the QUERY_STRING environment variable.
Read from STDIN.
Write output to STDOUT
Response is sent back to the client.
Response's body + metadata
Inside a CGI Program

Environment Variables in Perl

- Believe it or not…much much mcuh…easier than C…

```
#!/usr/bin/perl

foreach $i ( keys(%ENV) ) {
    print "$i : $ENV{$i}\n";
}
```

What is "%ENV"?

QUERY_STRING	=	……
REQUEST_METHOD	=	……
……	=	……

The keys

The values

[Example] read_env_ver1.cgi

Environment Variables in Perl

```
#!/usr/bin/perl -w

$method = $ENV{‘REQUEST_METHOD’};
if($method eq 'POST') {
    $length = $ENV{'CONTENT_LENGTH'};
    read(STDIN, $content, $length);
}
elsif($method eq 'GET') {
    $content = $ENV{'QUERY_STRING'};
}
else {
    print "Content-type: text/plain\n\n";
    print "Error!! Unknown method: $method\n";
    exit;
}
print $content, “\n”;
```

Similar to read() in C.

[Example] read_env_ver2.cgi

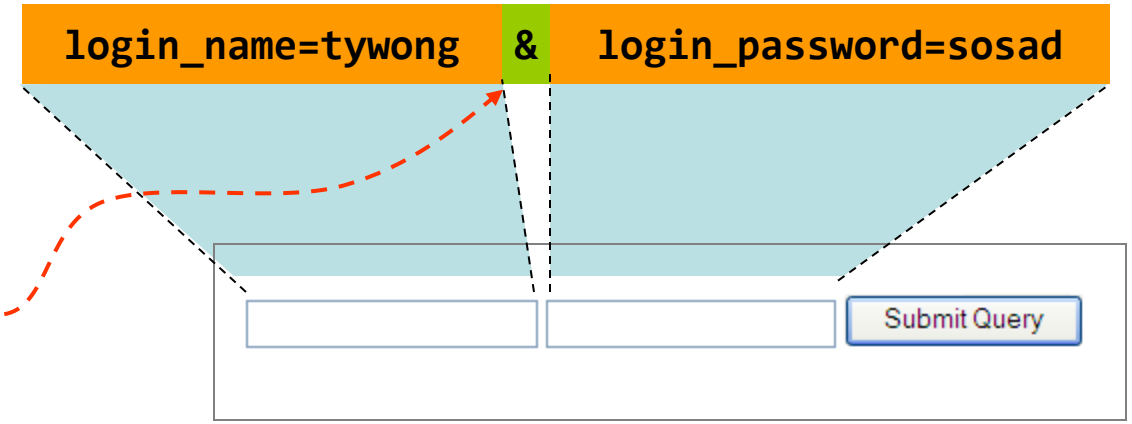
Extracting useful information…

- Pattern extraction?
  - Yes, Perl's true power…
  - The problem is how to write the regex?

Commonly stands for "regular expression".

```
@array = /(.+)&(.+)/
```

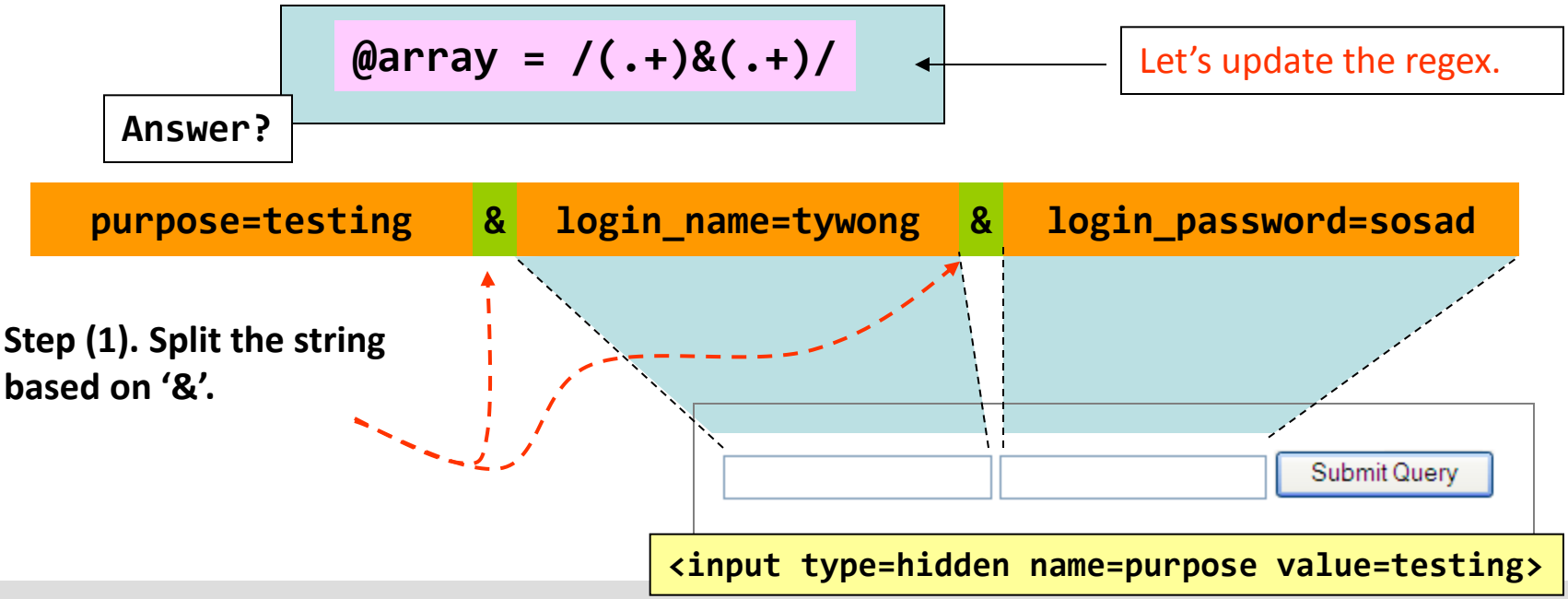
Answer?



Step (1). Split the string based on '&'.

Extracting useful information…

- Pattern extraction?
  - Yes, Perl's true power…
  - The problem is how to write the regex?



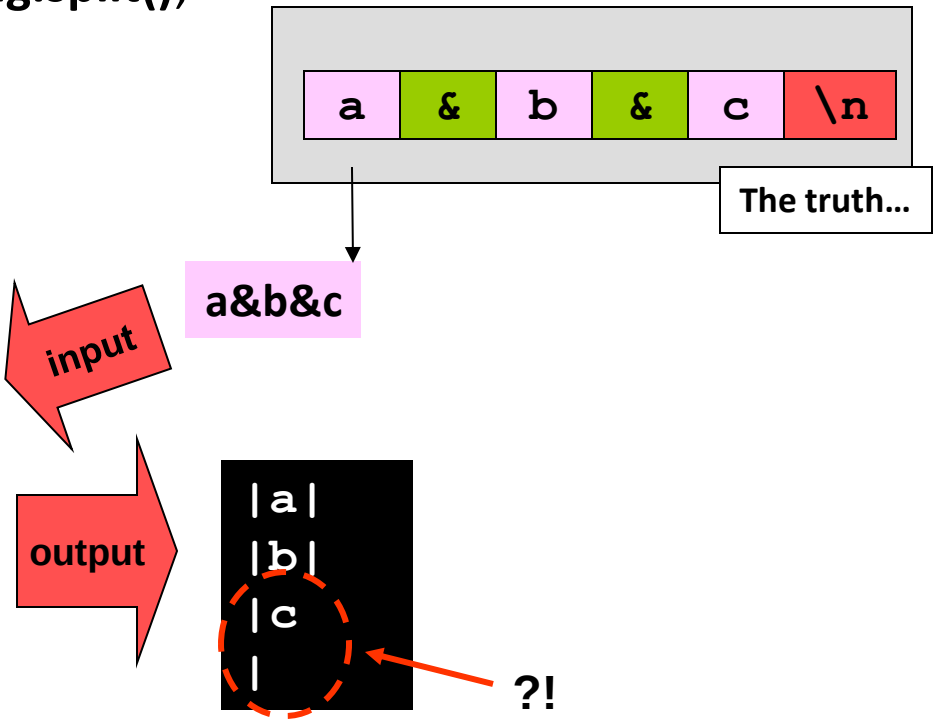
More on patterns…

- **split() & join()**…
  - **split()** splits a string into places based on a pattern…
  - Note: the concepts are so good that PHP and JavaScript.
    - JavaScript: **Array.join(), String.split()**;
    - PHP: **array_join(), split()**;

This can be any pattern, which is a regex.

```
$_ = <STDIN>;
@array = split( /&/ );

foreach $i (@array) {
    print "|$i|\n";
}
```

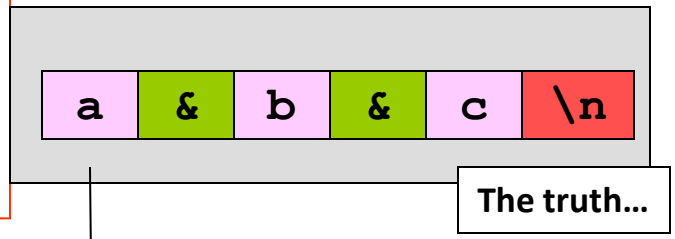


More on patterns…

chomp() is to remove one newline character at the end of the string.

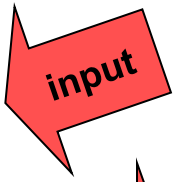
Another similar call is **chop()**, which remove one character at the end of the string.

chomp() is **very useful** and is commonly used when
- reading files line by line;
- reading from the **<textarea>**

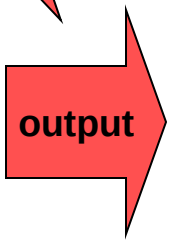



```
$_ = <STDIN>;
chomp($_);
@array = split( /&/ );

foreach $i (@array) {
    print "|$i|\n";
}
```



a&b&c



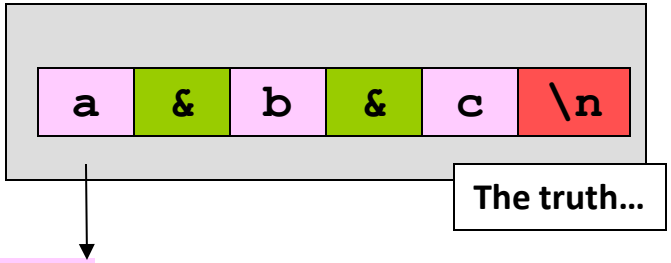

```
|a|
|b|
|c|
```

[Example] split.cgi

More on patterns…

Encore again!

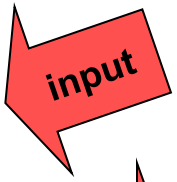
"There's more than one way to do it!" Perl motto!



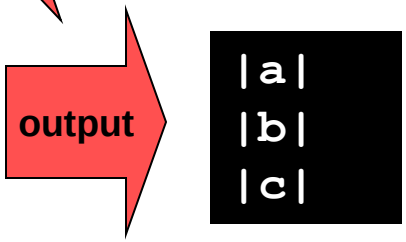
Another way to write the program…

```
$input = <STDIN>;
chomp($_);
@array = split( /&/, $input );

foreach $i (@array) {
    print "|$i|\n";
}
```



a&b&c

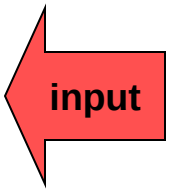


More on patterns…

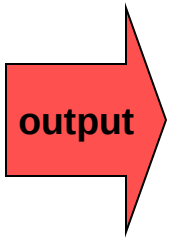
- **split() & join()**…
  - **join()** is the opposite to **split()**…
  - just give the call a string (not pattern) and an array, it will glue them together.

```
@array = ();
while($input = <STDIN>) {
    chomp($input);
    push(@array, $input);
}

$bigger_string = join(", ", @array);
print $bigger_string, "\n";
```




```
a
b
c
[Ctrl + D]
```

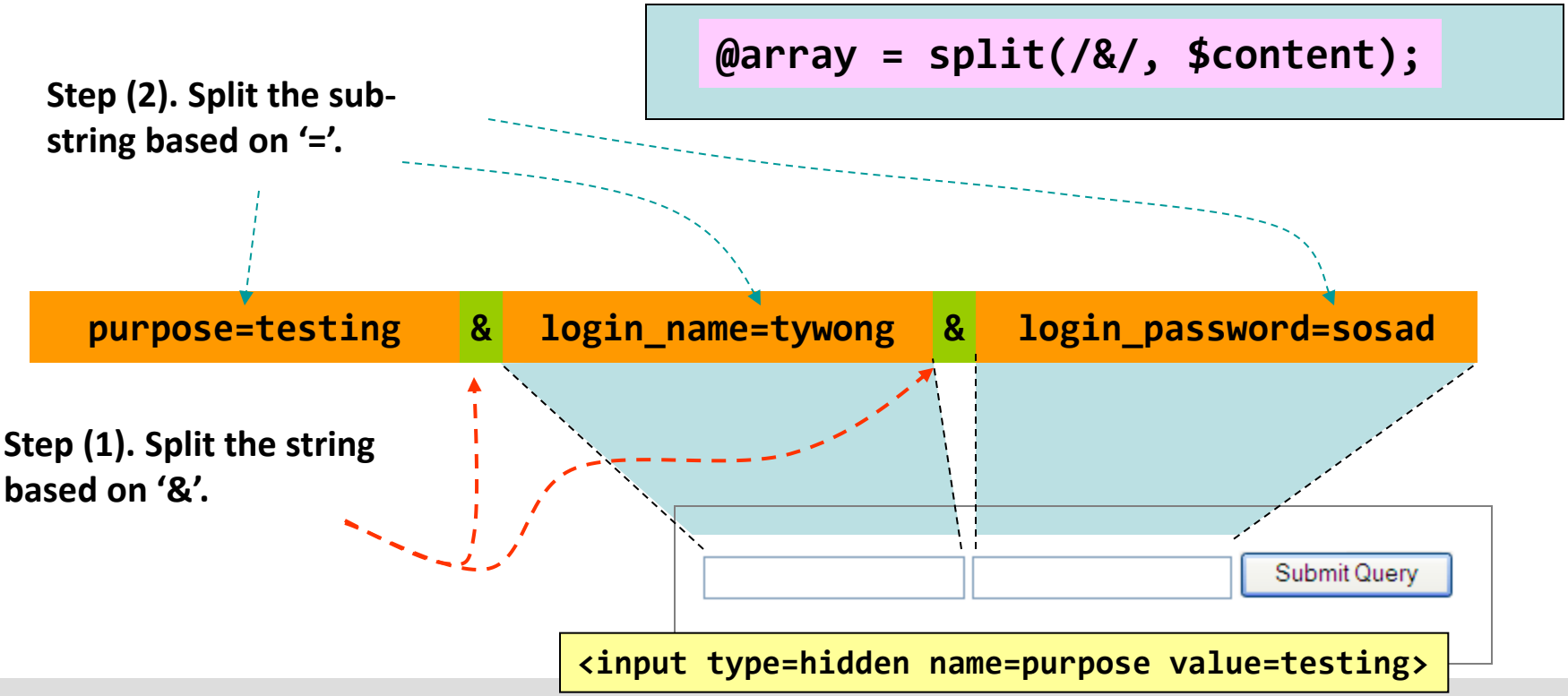



```
a, b, c
```

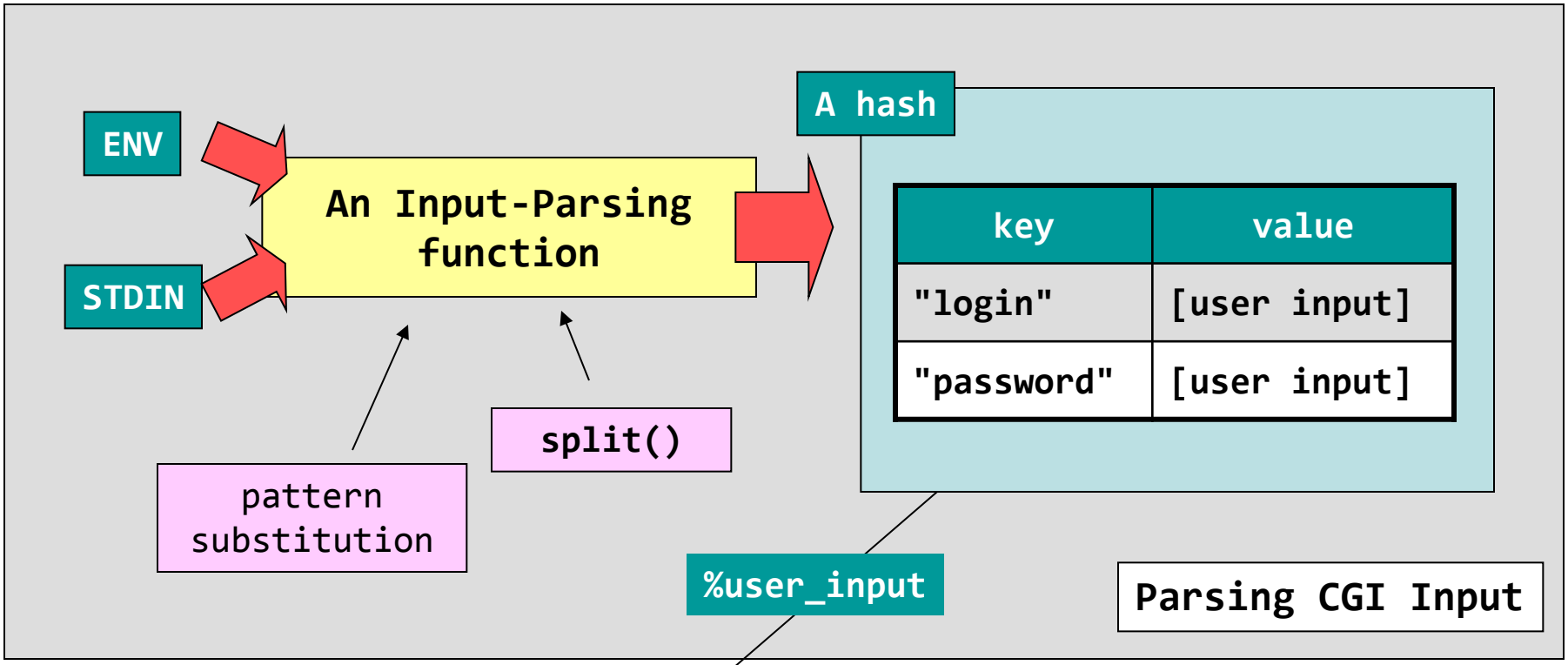
[Example] join.cgi

Extracting useful information (cont)

- Pattern extraction?
  - OK, let's use **`split()`**!



Extracting useful information (cont)




```
## Testing part ##
foreach $i (sort(keys(%user_input))) {
    print "|$i| = |$user_input{$i}|\n";
}
```

Print the list of inputs, sorted according to the keys.

Extracting useful information (cont)

Step (1)

```
#!/usr/bin/perl -w

# suppose the inputs are stored in $content...

@array = split(/&/, $content);
foreach $i (@array) {
    ($key, $value) = split(/=/, $i);
    $user_input{$key} =  $value;
}
```

Create an array of **(name, input)** pairs.

Separate the name and the input.

[Example] parseCGI_step1.cgi

Extracting useful information (cont)

Step (2)

```
#!/usr/bin/perl -w

# suppose the inputs are stored in $conter

@array = split(/&/, $content);
foreach $i (@array) {
    ($key, $value) = split(/=/, $i);
    $value =~ tr/+/ /;
    $user_input{$key} =  $value;
}
```

This line is to change '+' into ' '.

What is "`$value =~ tr/+/ /;`"?

equivalent

```
$_ = $value;
tr/+/ /;
$value = $_;
```

[Example] parseCGI_step2.cgi

Extracting useful information (cont)

Step (2)

```
$input = "chess";
$input =~ tr/s+/r/;
print $input, "\n";
```

→ **cherr**

```
$input = "chess";
$input =~ s/s+/r/;
print $input, "\n";
```

→ **cher**

```
$input = "chess";
$input =~ tr/s/r/;
print $input, "\n";
```

→ **cherr**

```
$input = "chess";
$input =~ s/s/r/;
print $input, "\n";
```

→ **chers**

Then, what is "`tr/+/ /;`"?

similar to … → `s/+/ /`

Two main differences…first:

The searching part for "**tr/**" is **a set of characters**.

But, for "**s/**", the searching part is **a regex**.

[Example] perl_tr_example.cgi

Extracting useful information (cont)

Step (2)

2 of the most useful matching modes for regex:

g	Replace globally, i.e. all occurrences
i	Do case-insensitive pattern matching.
…	…

```
$input = "chess";
$input =~ s/s/r/g;
print $input, "\n";
```

cherr

Then, what is "`tr/+/ /;`"?

similar to … → `s/+/ /`

Two main differences…first:

The searching part for "**tr/**" is **a set of characters**.

But, for "**s/**", the searching part is **a regex**.

[Example] perl_tr_example.cgi

Extracting useful information (done)

Step (3)

```
#!/usr/bin/perl -w

# suppose the inputs are stored in $content...

@array = split(/&/, $content);
foreach $i (@array) {
    ($key, $value) = split(/=/, $i);
    $value =~ tr/+/ /;
    $value =~ s/%(..)/chr(hex($1))/ge;
    $user_input{$key} =  $value;
}
```

Entire statement:
To replace %xx into an ASCII character.

e	execute any commands inside before substituting.

`%(..)` ← matching pattern.

`chr(hex($1))` ← substituting pattern.

What are **chr()** and **hex()**?

Let's visit "www.cpan.org"

[Example] parseCGI_step3.cgi

A very short exercise…

- Visit the following page:

  http://appsrv.cse.cuhk.edu.hk/~csci4140/cgi-bin/week02_challenge

- Design a set of substitution expressions such that…
  - **Requirement 1**:
    - Newline characters typed inside the textarea will be transformed into the newline tag `<br>` in the "Live Display" area.
  - **Requirement 2**:
    - The CGI initially allows users to insert HTML tags. Now, we want to disable such a feature. How?

Summary

- It is always ugly to have one function only...

```
$method = $ENV{‘REQUEST_METHOD’};
if($method eq 'POST') {
    $length = $ENV{'CONTENT_LENGTH'};
    read(STDIN, $content, $length);
}
elsif($method eq 'GET') {
    $content = $ENV{'QUERY_STRING'};
}
else {
    print "Content-type: text/plain\n\n";
    print "Error!! Unknown method: $method\n";
    exit;
}
```

Retrieve input from user.

$content

```
@array = split(/&/, $content);
foreach $i (@array) {
    ($key, $value) = split(/=/, $i);
    $value =~ tr/+/ /;
    $value =~ s/%(..)/chr(hex($1))/ge;
    $user_input{$key} =  $value;
}
```

Parse input from user.

Next, We quickly cover how to:

- write subroutines, and
- pass parameters.

Program codes for subroutine

all_files.zip	funny_name.cgi	funny_name_use_strict.cgi	hash2array.cgi
pass_array_correct.cgi	pass_array_problem.cgi	pass_by_XX.cgi	pass_by_value.cgi
pass_hash.cgi	pass_scalar.cgi	ref_array.cgi	ref_hash.cgi
ref_scalar.cgi	scope1.cgi	scope1_with_my.cgi	scope2.cgi
scope2_with_my.cgi	scope2_with_strict.cgi	---	---

Fall 2011, CSCI4140, Department of Computer Science and Engineering, The Chinese University of Hong Kong.

http://appsrv.cse.cuhk.edu.hk/~csci4140/cgi-bin/perl/script/subroutine/

Subroutines & Variable Scoping

*- using '**my**' and '**strict**'...*

"**Funny**" variable names…

- Execute the following program, what will you get?

```
$hello = "Say hello to you";
print "The message is: '$hallo'\n";
```

```
$ perl funny_name.cgi
The message is: ''
$ _
```

Say what?!

Perl allows you to declare or initialize variables **at any locations**.

[Example] funny_name.cgi

Misspelt variable names…

- To fix this "*bug*"…

```
use strict;

my $hello = "Say hello to you";
print "The message is: '$hallo'\n";
```

"**use strict**" is to enforced variable declaration before using.

"**my**" is to declare a name as a local variable.

```
$ perl funny_name_use_strict.cgi
Global symbol "$hallo" ......
Execution of funny_name_use_strict.cgi aborted ......
$ _
```

Now, the error can be spotted!

[Example] funny_name_use_strict.cgi

Other uses of "**my**"… example #1

```
$hello = "outside";

sub foobar
{
    $hello = "inside";
    print "$hello\n";
}

print "$hello\n";
foobar();
print "$hello\n";
```

This acts as a global variable from the point of view of the subroutine "**foobar()**".

To define the subroutine "**foobar()**".

```
$ perl scope1.cgi
outside
inside
inside
$ _
```

[Example] scope1.cgi

Other uses of "**my**"… example #1

```
$hello = "outside";

sub foobar
{
    my $hello = "inside";
    print "$hello\n";
}

print "$hello\n";
foobar();
print "$hello\n";
```

The keyword "**my**" defines and limits a name inside the subroutine "**foobar()**".

```
$ perl scope1_with_my.cgi
outside
inside
outside
$ _
```

[Example] scope1_with_my.cgi

Other uses of "**my**"… example #2

```
#$hello = "outside";

sub foobar
{
    $hello = "inside";
    print "$hello\n";
}

print "$hello\n";
foobar();
print "$hello\n";
```

```
$ perl scope1_with_my.cgi

inside
inside
$ _
```

Say What?!

Perl has a very flexible, global scoping concept.

[Example] scope2.cgi

Other uses of "**my**"… example #2

```
#$hello = "outside";

sub foobar
{
    my $hello = "inside";
    print "$hello\n";
}

print "$hello\n";
foobar();
print "$hello\n";
```

Again, using the keyword "**my**" can save you days and nights of crazy debugging…

```
$ perl scope1_with_my.cgi

inside

$ _
```

[Example] scope2_with_my.cgi

Other uses of "**my**" & "**strict**"… example #2

```
use strict;

#$hello = "outside";

sub foobar
{
    my $hello = "inside";
    print "$hello\n";
}

print "$hello\n";
foobar();
print "$hello\n";
```

Important: using "**use strict**" is the safest measure!

```
$ perl scope2_with_strict.cgi
Global symbol "$hello" ......
......
$ _
```

[Example] scope2_with_strict.cgi

Parameter Passing

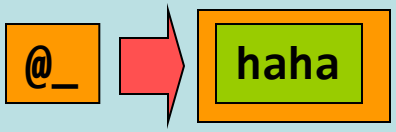
- Passing Scalar, Array, & Hash...

- Pass-by-value or Pass-by-reference?

Passing Scalar to Subroutines...

Parameter passing in Perl is "**FUNNY**"!

All the arguments are packed into an array. Such an array can be referenced inside the subroutine by the name "**@_**".




```
sub foobar {
    $input = shift @_;
    if($input) {
        print "Input = '$input'.\n";
    }
    else {
        print "No input.\n";
    }
}

foobar();
foobar("haha");
```

```
$input = shift @_;
```

So, this statement reads the first argument out.

```
$ perl pass_scalar.cgi
No input.
Input = 'haha'.
$ _
```

[Example] pass_scalar.cgi

Passing Array to Subroutines

```
sub foobar {
    my @array = shift;
    foreach my $i (@array) {
        print "$i\n";
    }
}

my @array = ("hello", "world");
foobar(@array);
```

Hey! Where is the "WORLD"?!

```
$ perl passing_array.cgi
hello
$ _
```

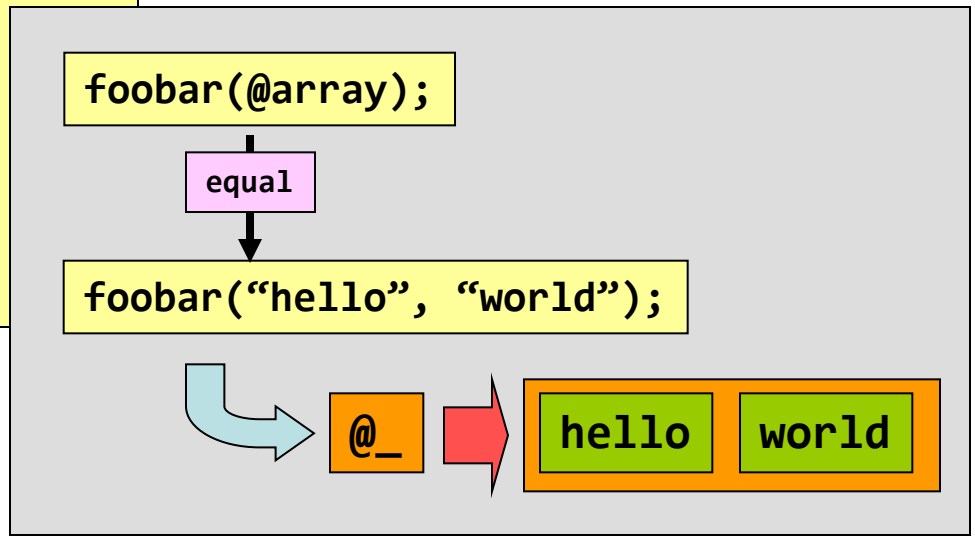
[Example] pass_array_problem.cgi

Passing Array to Subroutines

```
sub foobar {
    my @array = shift;
    foreach my $i (@array) {
        print "$i\n";
    }
}

my @array = ("hello", "world");
foobar(@array);
```

But, "**shift**" can only give you a scalar...not an array...



[Example] pass_array_problem.cgi

Passing Array to Subroutines

- Please reconstruct the array by yourself...

```
sub foobar {
    my @array = ();
    foreach my $i (@_) {
        push(@array, $i);
    }
    foreach my $i (@array) {
        print "$i\n";
    }
}

my @array = ("hello", "world");
foobar(@array);
```

```
sub foobar {
    my @array = @_;

    foreach my $i (@array) {
        print "$i\n";
    }
}
```

If there is only one argument and that argument is an array, then…

[Example] pass_array_correct.cgi

Passing Hash to Subroutines?

- How to read an argument of type “hash”?

```
use strict;

sub foobar {

        ?

}

my %hash = (
    "a" => "Say",
    "b" => "Hello",
    "c" => "World",
    "d" => "Together");
foobar(%hash);
```

First, we have to understand what the hash really is...

```
use strict;

my %hash =
    ("msg" => "hello world");
my @array = %hash;

foreach my $i (@array) {
        print "$i\n";
}
```

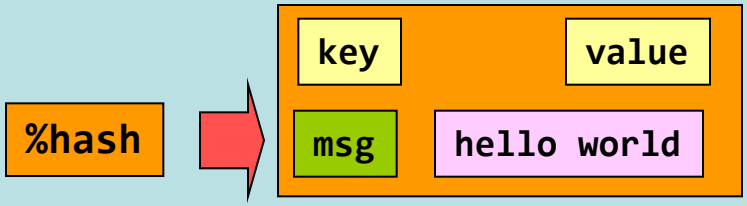
[Example] hash2array.cgi

Passing Hash to Subroutines?

- How to read an argument of type “hash”?

Well…the truth is that there is no hash; **a hash is just an array**.

Elements with even indices are keys; elements with odd indices are values;




```
$ perl hash2array.cgi
msg
hello world
$ _
```

First, we have to understand what the hash really is...

```
use strict;

my %hash =
    ("msg" => "hello world");
my @array = %hash;

foreach my $i (@array) {
        print "$i\n";
}
```

[Example] hash2array.cgi

Passing Hash to Subroutines?

- Re-construct the hash in the subroutine…

Have to ensure that the argument is a hash. Otherwise…

```
use strict;

sub foobar {

        ?

}

my %hash = (
    "a" => "Say",
    "b" => "Hello",
    "c" => "World",
    "d" => "Together");
foobar(%hash);
```

```
my %hash;
while( my $key = shift ) {
    $hash{$key} = shift;
}

foreach my $i (sort(keys(%hash))) {
    print "$i \t $hash{$i} \n";
}
```

Re-create the hash by yourself…

[Example] pass_hash.cgi

Parameter Passing

~~- *Passing Scalar, Array, & Hash...*~~

- *Pass-by-value or Pass-by-reference?*

Pass By Value or Reference?

- **A quick test**. According to previous examples, we can sense that...**Perl implements pass by value**.

```
sub foobar {
    $msg = shift;
    $msg = "hell";
}

$msg = "hello";
foobar($msg);
print "$msg\n";
```

```
$ perl pass_by_XX.cgi
hell
$ _
```

Say what? This is a pass-by-reference behavior. Can you explain?!

[Example] pass_by_XX.cgi

Pass By Value or Reference?

- **A quick test**. According to previous examples, we can sense that...**Perl implements pass by value**.

```
sub foobar {
    my $msg = shift;
    $msg = "hell";
}

$msg = "hello";
foobar($msg);
print "$msg\n";
```

```
$ perl pass_by_value.cgi
hello
$ _
```

What a relieve....this is pass by value.

[Example] pass_by_value.cgi

Pass By Reference – Scalar

```
$ perl ref_scalar.cgi
hell
$ _
```

```
sub foobar {
    my $ptr = shift;
    $$ptr = "hell";
}

$msg = "hello";
foobar(\$msg);
print "$msg\n";
```

A reference is treated as a scalar.

$$ptr – it means dereferencing the "pointer" value "**$ptr**" as a scalar variable.

\$msg – it represents the reference to the variable "**$msg**".

[Example] ref_scalar.cgi

Pass By Reference – Array

```
use strict;

sub foobar {
    my $ptr = shift;
    my $n = scalar @$ptr;
    for(my $i = 0; $i < $n; $i++) {
        chop( $$ptr[$i] );
    }
}

my @msg = ("hello", "world");
foobar(\@msg);
foreach my $i (@msg) {
    print "$i\n";
}
```

"**@$ptr**" – it means dereferencing the "pointer" value "**$ptr**" **as an array variable**.

[Example] ref_array.cgi

Pass By Reference – Hash

"**%$ptr**" – it means dereferencing the "pointer" value "**$ptr**" **as a hash variable**.

```
use strict;

sub foobar {
    my $ptr = shift;
    foreach my $i (keys(%$ptr)) {
        chop( $$ptr[$i] );
    }
}

my %msg = ("a" => "hello", "b" => "world");
foobar(\%msg);
foreach my $i (keys(%msg)) {
    print "|$i|=>|$msg{$i}|\n";
}
```

[Example] ref_hash.cgi

Let’s move to real CGI scripts!

- Now, we have equipped ourselves the skill to write bigger Perl programs....
  - E.g., Web login and logout mechanism.

- By the way, Perl module is a set of functions that form a library.
  - E.g., The CGI module implements all the CGI handlings.
  - We are not going to teach it...but you’re welcome to learn from:

http://world.std.com/~swmcd/steven/perl/module_mechanics.html