Introduction to Perl Programming

H4311S B.00

Module 1

What Is Perl?

- Perl — Practical Extraction and Reporting Language.
- Perl has similarities with the C programming language.
- Perl has similarities with shell scripting.
- Perl is a linear programming language, not a cyclic processor like `sed` and `awk`.
- Perl has built in commands and functions.
- Perl uses modules to extend its capabilities.
- Extensive documentation is available on the Comprehensive Perl Archive Network (CPAN).

Parts of a Perl Program


```
1. #! /opt/perl5/bin/perl
2. # @(#) Version A.00
3. # $Revision: 1.2$
4. #
5. # This program prints a greeting then exits.
6. #
7. print "Hello, world.\n"; # tradition
8. print "Welcome to Perl programming.\n";
9.
10.exit;
```

Creating a Perl Program

- Plan the flow of the program.
- Use a text editor to create a program file.
- Specify in the first line that Perl will be the interpreter.
- Add a line for version control.
- Use comments to document the program.
- Execute the program:

  Make it executable: `chmod +x prog.pl`

  or

  Execute as input to Perl: `perl prog.pl`

Perl Statements

H4311S B.00
Module 2

Format of Perl Statements

- A simple statement consists of
  - a command or subroutine call
  - an assignment
  - a terminating semicolon
- A compound statement consists of
  - a condition
  - a block
- A block consists of
  - a pair of braces
  - a set of simple statements

Statements — Example


```
1    #! /opt/perl5/bin/perl
2    #
3    #  @(#) statements.pl: Version 1.0
4    # This program demonstrates Perl statements
5    #
6    $var1 = "I'm in the outer block.";
7    print  "$var1\n";
8    {
9        my $var1 = "I'm in the inner block.";
10      print "$var1\n";
11   }
12   print "$var1\n";
13   exit;
```

Variables

- Perl does not have data types.
- Perl will store data in
  - scalar variables
  - lists
  - arrays
  - hashes
- Perl will convert data to the proper type for the statement.
- Scalar variables
  - start with `$` followed by alpha followed by an alphanumeric
  - allow underscores
  - store a single value that may contain white space

Scalar Variables

- Assigning scalar variables:

```
$cost = 50000;
$margin = 0.15;
$product = "car";
```

- Using scalar variables:

```
$price = $cost + ($cost * $margin);
$desc = "A red car with lots of extras. Only $price
dollars";
print ("The cost is: ", $cost);
```

Commands

- Built-in commands for
  - variable manipulation
  - input and output
  - program flow
  - management of processes, users, groups
  - network information
  - IPC and sockets

- User-defined subroutines

Evaluation and Assignment

- Has the format

  operand1 operator operand2

- Operand1 can be a literal or a variable in some expressions.
- Operand1 must be a variable if the operator is an assignment.
- Operand2 can be a literal, a variable, or the return value of a function call.
- The operator can be any one of the operators in the list of operators.

Operators

- list operators, parentheses, braces, quotes, — `()`{`}`' "
- array and hash index — `[]` {}
- dereference and method calls — `->`
- increment, decrement — `--` `++`
- unary operators — `+` `-` `!` `~`
- arithmetic operators — `+` `*` `/` `%` `-`
- bit operators — `&` `|` `^`
- relational — `gt` `>` `<` `eq`
- logical — `&&` `||`
- comma — `,`
- logical — `and` `or` `xor`

Managing Data

H4311S B.00
Module 3

Standard File Descriptors

- Three file descriptors are opened automatically:
  - STDIN the standard input device (the keyboard)
  - STDOUT the standard output device (the monitor)
  - STDERR the standard error device (the monitor)
- Some commands will use them as the defaults:
  - `<>` is the same as `<STDIN>`

```
print "This is a line of output.\n"
print STDOUT "This is a line of output.\n"
```

- `STDERR` must be specified explicitly
  - `print STDERR "This is an error!\n"`

Opening Files

open (*filehandle*, "`mode filename`");

close *filehandle*;

- Filehandle is any name you want to use.
- Mode can be the following:
  - omitted or `<` input (reading)
  - `>` output (writing)
  - `>>` append (writing)
  - `+>` input and output truncate the file if it exists.
  - `+<` input and output do not truncate the file.
- Filename in quotes is the pathname of the file:
  - `filename` alone is a file.
  - `| filename` is a command that reads from the pipe.
  - `filename |` is a command that writes to the pipe.

Reading and Writing Files

Reading

- `$var = <FILEHANDLE>`
- `read`
- `getc`
- `seek`
- `tell`
- `eof`

Writing

- `print`
- `printf`
- `write`
- `formats`
- `syswrite`

`print` and `printf`

- `print` outputs a list of items that may be enclosed in parentheses.

```
print "The value of var is ", $var, ".\n";
print($var+7, " is more than ", $var-7, ".\n");
```

- `printf` outputs a formatted string.
  - `printf`("format string", positional parameter);
  - The format string contains literals and field specifiers that will be replaced by the positional parameters.

```
printf "The value of var is %s.\n", $var;
printf "%d is more than %d.\n", $var+7, $var-7;
```

- Substitutions and evaluations will be done before the data is output.

`write` and Formats

- `write` sends output to the filehandle specified using its associated
- Filehandles start with a format name that matches the filehandle name
- The default file handle is `STDOUT`.
- The select function allows the default filehandle to be changed.
- The format name is assigned to default filehandles using the `$~` special variable.
- The default format can be changed by assigning a format name to the special variable:

```
select NEWDEFAULT;
$~ = "NEWFORMAT";
```

Formats

```
1. format SALARYFORM =
2. Employee              Salary
3. ========================
4. @<<<<<<<<<<     @>>>>>>>
5. $name   ,       $salary
6. .
7. $~ = "SALARYFORM";
8. $name = "M Mouse";
9. $salary = 1000;
10.write;
```

Looping and Branching

H4311S B.00
Module 4

What Is True? False?

- In Perl, particular values are considered FALSE
  - Numeric: 0, 0.0
  - String: “”, “0”
  - Other: undef, null
- Everything else is TRUE!
  - 1, “hello”, 3.1415926, -32, 0x0003152BF0, “0.0”, …
- Commands
  - return value of zero or null is FALSE
    - e.g. int(“0.0”)
  - return value of non-zero or non null is TRUE

if

- ```
  if (condition)
  {
      block;
  }
  ```

- ```
  if (condition)
  {
      block;
  }
  else
  {
      block;
  }
  ```

- ```
  if (condition)
  {
      block;
  }
  elsif (condition)
  {
      block;
  }
  .
  .
  .
  else
  {
      block;
  }
  ```
  ```

unless

- ```
  unless (condition)
  {
      block;
  }
  ```

- ```
  unless (condition)
  {
      block;
  }
  else
  {
      block;
  }
  ```

`while` Loop

- ```
  while (condition)
  {
      block;
  }
  ```

- executes if condition is true
- will execute 0 or more times
- stops executing when (condition) is false
  ```

`until` Loop

- ```
  until (condition)
  {
      block;
  }
  ```

- will execute 0 or more times
- executes if condition is false
- stops execution when condition is true
  ```

`for` Loop

- ```
  for (initializer; condition; iterator)
  {
     block
  }
  ```
- `initializer` can be any valid Perl expression, but is usually a single assignment statement
- `condition` is a relational or conditional expression to evaluate
- `iterator` is executed at the end of each block i.e. just before the next iteration
- ```
  for ($i = 0; $i < 10; $i++)
  {
          print $i;
      }
  ```

foreach Loop

```
foreach  $value (list)
{
    block;
}
```

- executes the command block once for each element of the list
- stops execution when no more elements are in the list

Lists, Arrays, and Hashes

H4311S B.00
Module 5

Lists

- A list is an ordered set of values enclosed in parentheses.
- A list has no name.
- Each element in the list can be accessed by an index.
- The index is enclosed in square brackets.
- The members of a list can be literals, scalars, or other lists.
- A list can be used as an *rvalue* or an *lvalue*.

```
(1, 2, 3, 4, $varX)              – a list
($v1, $v2) = <STDIN>             – a list as an lvalue
($v1, $v2, $v3) = (1, 2, 3)      – a list as an rvalue
```

Working with Lists


```
1.   (1, 2, 3)
2.   (1..10)
3.   (a..z,A..Z)
4.   (1, 2, (a, b, c), 5, (3, 4))
5.   ($item, $cost) = (“lunch”, 10.00)
6.   ($item, $cost) = ($description, $price, $part_number)
7.   ($animal) = qw(cat dog fish bird)
8.   $animal = qw(cat dog fish bird)
9.   $animal = (“cat”, “dog”, “fish”, “bird”)[0]
10.  $sz = (stat(inventory.db))[7]
11.  $word = (“cat”, “dog”, “fish”, “bird”)[rand(3)]
```

List Related Commands

- `join` join list elements together into a single string
- `split` create a list by splitting up a string
- `reverse` reverse a list
- `sort` sort a list
- `map` perform activity for each element of a list

Arrays

- An array is a list with a name.
- The name must start with `@`.
- The name starts with a letter followed by alphanumeric and underscore.
- An array can be populated from a list
  ```
  @arrayname = (list);
  ```
- Array elements are accessed using an `index`.
  ```
  $element = $arrayname[index];
  ```
- `index` can be any expression that evaluates to a number.
- A slice is an array that is a subset of a larger array.
  ```
  @array_slice = @array[index list];
  ```

Working with Arrays

```
1.  @animals = (“cat”, “dog”, “fish”, “bird”)
2.  $animals[1]
3.  $animals[-1]
4.  @numbers = (1..10)
5.  @nums = @numbers
6.  ($my_pet, $your_pet) = @animals
7.  ($my_pet, @rest_of_the_animals) = @animals
8.  ($your_pet, @rest_of_the_animals) = @rest_of_the_animals
9.  print (@animals)
10. $array_size = @animals
11. $animals[5] = “lizard”
12. @slice = @animals[1, 3]
13. @slice = @animals[0..2]
```

Array Related Commands

- `pop` remove and return last array element
- `push` append a list to an array
- `shift` remove and return first array element
- `unshift` prepend a list to an array
- `splice` insert a list into an array

Hashes

- A hash is a named list that contains key-value pairs
- The key is frequently a string
- The name starts with a `%`.
- The first character is a letter, followed by alphanumeric or underscore.
- Hashes may be populated from a list

```
%hash = (key1, value1, key2, value2,…);
```

or

```
%hash = (key1 => value1, key2 => value2,…);
```

- Access a value by specifying its key

```
$hash{key2};
```

Working with Hashes

```
1. %animals=(cat,persian,dog,collie,bird,eagle);
2. %animals= (
        cat => persian,
        dog => collie,
        bird=> eagle
   );

3. $animals{“fish”}  = “shark”;

4. %new_animals = %animals;
```

Hash Related Commands

- `exists` check if a key is in the hash
- `keys` list all of the keys in the hash
- `values` list all of the values in the hash
- `delete` delete a key-value pair or pairs
- `each` list the next key-value pair

Looping and Branching Controls

H4311S B.00

Module 6

Modifiers

- Any statement may be augmented with a modifier:
  - *statement* `modifier (`*condition*`);`
  - `if`, `unless`, `while`, `until` and `foreach`
- The condition is evaluated before executing the statement
- `if` and `unless` cause the statement to be executed once, or not at all, depending on the condition.
- `while`, and `until` cause the statement to be executed 0 or more times, depending on the condition
- **Exception**: if the statement is a `do` statement modified with `while` or `until`, the condition is checked after the statement is executed. Thus a `do` statement will be executed at least once with `while` and `until`.

Using “Short-Circuit” Statements

- Perl will not evaluate the right hand side of a logical operator if it would not change the true / false result:
  - For logical `and`, and `&&` if the left side is `false`, the result is `false` no matter what the right side is
  - For logical `or`, and `||`, if the left side is `true`, the result is `true` no matter what the right side is
- The result of a command can be considered `true` or `false`.
- Consequently, two commands can be connected with `or` or `and`, and the second command will be executed conditionally:
  - `open FH, “< $fname” or die “Could not open $fname”`
  - `($name) and print “Name is $name\n”;`

Modifying Execution of a Loop

- `next` — ends the current execution of the loop and resumes executing at the condition.
- `last` — ends the current execution of the loop and resumes execution after the current block.
- `redo` — ends execution of the current loop and resumes execution after the condition.
- Instead of the default, you can specify a label:

```
next LABEL;
last LABEL;
redo LABEL:
```

- If `LABEL` is omitted, `next`, `last` and `redo` use the current block.

Labels

- A label provides a name by which a block of code can be referenced.
- Labels can be used by `redo`, `last`, `next`, and `goto`.
- A label consists of an alpha or underscore, followed by one or more alphanumeric or underscore characters.
- A label is terminated by a colon.
- A label is case sensitive.

Pattern Matching

H4311S B.00
Module 7

Pattern Matching

- Pattern Matching is part of the Perl language, not an add-on
- Pattern Matching uses Binding Operators, Regular Expressions (REs), Commands, and Command Modifiers
- Binding operators associate a string “topic” to a RE “pattern”

```
"Sail Away" =~  m/^Sail \w+/i;
```

- REs express patterns using literals, and special characters
- Commands specify how the pattern is used against the bound topic: `m//` (match), `s///` (substitute), `tr///` or `y///` (transliterate)
- Command Modifiers change command behaviour

  `i` (ignore case), `g` (global), `s` (squash)

Uses for Pattern Matching

- Verify a string/topic matches a pattern — returns true or false.

```
if ($line =~ /^root:/) # m assumed: m/^root:/
```

- Save whether, or what, the RE pattern matched in the topic.

```
$matched = $line =~ m/RE/;  # matched = 1/0
@matches = $line =~ m/RE/g; # saves matches
```

- Perform substitution or translation on the string

```
$line =~ s/RE/string/g;
$line =~ tr/string1/string2/;
```

- Extract parts of the topic without changing it:

```
$name, $host, $domain) = 'phil@sailing.hp.com' =~ /(\w+)@(\w+)\.(.*)/;
```

Binding with the m// Command

- Binding (=~) associates a string “topic” with a Regular Expression “pattern”
- The `m` match command indicates whether or not the topic matches
  - a `0` (no match) or `1` (match) is returned if binding in *scalar* context
  - a `()` (no match) or `(1)` (match) is returned if binding in *list* context

```
$a   = “Abe Lincoln” =~ m/Wash/;     # $a is 0
@arr = “Abe Lincoln” =~ m/Lincoln/; # @arr is (1)
```

- The `m` is assumed if missing

```
@arr = “Abe Lincoln” =~ /Wash/;# @arr is ()
```

- Topic and binding may be omitted: if so, `$_` is bound

```
$_ = “Abe Lincoln”;
$a = /Lincoln/;  # $a is 1
$character = (/Lincoln/) ? “honest” : “cagey”;
```

What Is a Regular Expression?

- A Regular Expression (RE) is
  - a pattern of what to look for in a string, usually delimited with /
  - interpolated before processing, just like a double-quoted string
  - used with `m//` and `s///` commands, as well as with Perl functions (e.g. `split`)
- Regular Expressions can contain any mix of
  - literal characters
    ```
    /root/, /42/, /# Done!/
    ```
  - special characters (“metacharacters”)
    ```
    /^root/, /[a-zA-Z]+/, /(0x)?[0-9a-fA-F]+/
    ```
  - metasymbols
    ```
    /\d/, /\w+\s\d+/
    ```

Literal Matching

- Most characters in an RE are matched to themselves:

  `yes:  /` matches "`yes:  45`", and "`ayes:  36`"

- Some characters have special meaning:

  `\  |  (  )  [  {  ^  $  .  * + ?`

- Precede special characters with backslash (`\`) to match them literally

  `/hp.com/` matches "`hp.com`" and "`hpicom`"
  `/hp\.com/` matches "`hp.com`", but not "`hpicom`"

- The delimiter is special, but may be changed:

```
m/\/usr\/tmp/    # matches /usr/tmp
m#/usr/tmp#      # same, but easier to read
```

  Note: the `m` is required when specifying a different delimiter than `/`

Special Characters

- `^,$` Anchors to the start, end of a line (or string)
- `[ ]` Matches one of the specified group of characters
- `.` Matches any single character (except newline)
- `\` Treat next character as literal; also, start metasymbol sequence
- `|` Separates alternatives
- `*` Matches 0 or more of the preceding RE element
- `+` Matches 1 or more of the preceding RE element
- `?` Matches 0 or 1 of the preceding RE element; also, create a minimal match for the preceding quantifier
- `{}` Used to specify quantifiers
- `()` Used to capture sub-expressions

Metasymbols

- A metasymbol is a character sequence with a special meaning
  - The sequence is not matched literally
  - The first character is `\`
- Specifying a specific, perhaps non-printable, character:

  `\a, \n, \r, \t, \f, \e, \007, \x07, \cx`
- Specifying one of a certain type of character:

  `\d,\D, \w,\W, \s,\S, \l, \u`
- Specifying an assertion / anchor / boundary:

  `\b, \B, \A, \Z, \z, \G`
- Start / End specified case of letters

  `\L, \U, \E`

Match a Single Character from a Group

- Use a period (`.`) to match any single character (except newline)

  `/c.t/` matches "`cat`", "`c t`", "`c.t`", "`boycott`"

- Use metasymbols to match pre-defined lists of characters
  - `\d` (digit) `\s` (white space) `\w` (word character)
  - `\D` (non-digit) `\S` (non-white) `\W` (non-word)
- Use `[`, `]`, – and `^` to specify a list of alternative characters
  - order doesn't matter (except for readability!)
  - ranges are specified using –

    `[abcde]`, `[ebdca]`, `[a-e]` # equivalent
  - `^`, when first, means "except for"; when not first, it means itself

    `[^0-9]`, `[a-z\-0-9]`, `[ABC^,_]`
  - Backslash and metasymbols may also be used: `[ \t]`

Character Matching Quiz

- Given the following list:
  a, abcd, ab9Cd, aBC, Abc, Abc1, Abc12a, .0901, Abcf, abc, abc2 ,
  bbbb, ABc, Abc3, bbabb, 99.99, 123

- Construct an RE, which matches words that:
  1. contain "abc“
  2. contain a number
  3. contain digits higher than 2
  4. has a b or B followed by a digit followed by a c
  5. has a 1 , 2 followed by a lower case letter

Anchors

- `^` anchors the pattern to the start of the string or a newline.
- `$` anchors the pattern to the end of a string or a newline.
- `\b` anchors to a word boundary.
- `\B` anchors to a non-word boundary.
- `\A` anchors to the start of a string.
- `\Z` anchors to the end of a string or a newline at the end.
- `\z` anchors to the end of a string.
- `\G` anchors to where the previous `m/RE/g` finished.

`/^root/` matches "`root`" and "`rooter`", but not "`chroot`"

`/root$/` matches "`root`" and "`chroot`", but not "`rooter`"

`/^root$/` matches only "`root`"

Quantifiers { }

- Quantifiers specify how many times a pattern should occur:
  - {1,6} at least once but no more than six
  - {3,3} or {3} exactly 3 times
  - {3,} a minimum of three times
- `*` match the preceding character **0** or **more** times
  `/do*r/` matches `dr dor door dooor`....
- `+` match the preceding character **1** or **more** times
  `/do+r/` matches `dor door dooor`....
- `?` match the preceding character **0** or **1** times
  `/do?r/` matches `dr dor`
- Default is ***maximal*** match; follow with ? for a ***minimal*** match:
  `*?, +?, ?? {}?` Makes the match minimal.

Anchors and Quantifiers — Examples

- `$string =` “This is a string that has words, sentences, and punctuation. It also has a newline embedded.
  So there it is. `\nStrings` like being, bekeeeping and bookkeeping are sometimes included.”

- Create a regular expression to locate:
  1. A line starting with capital S up to a word boundary.
  2. Repeat with a minimal match to return just the word.
  3. Match words with strings of 2 or more letter e’s.
  4. Match words that have only one letter e.

Saving Matched Data

1. Bind in a list context, and use the /g modifier

```
$str = "This is too risky.";
@arr = $str =~ /.is/g;
        # @arr gets ("his", " is", "ris")
```

2. Place parentheses around or within the pattern, and bind in a list context

```
@arr = $str =~ /(.is)/;    # @arr gets ("his")
@arr = $str =~ /((.)is)/; # @arr gets ("his", "h");
```

3. In scalar or list context, use parentheses to ***capture***, and ***backreferences*** to refer to them

```
"Abraham Lincoln" =~ /((\w+) (\w+))/;
# $1 = "Abraham Lincoln"
# $2 = "Abraham"
# $3 = "Lincoln"
```

Modifying Strings with s///

- Bind with the `s` substitute command to change the topic string

```
$topic =~ s/pattern/replacement/
```

- The operation counts the number of substitutions made

```
$str = "This is risky.";
$res = $str =~ s/.is/at/;  # $res is 1
print "$str\n"; # prints "Tat is risky."
```

- Use `/g` to replace globally

```
$str = "This is risky.";
@res = $str =~ s/.is/at/g;  # @res is (3)
print "$str\n"; # prints "Tatat atky."
```

More on Capturing and Backreferences

- Two forms of backreferences are available:
  - $1, $2, $3, ... persist until the next pattern match (`m//` or `s//`) completes
  - \1, \2, \3, ... persist only during the current binding
- In subsequent statements, use $N
- In substitutions and matching patterns, use \N

```
"jub-jub" =~ m/(\w+)-\1/;   # matches
"dim-sum" =~ m/\(\w+)-\1/; # doesn't match
```

- In substitution replacements, use either $N or \N:

```
$name = "Abraham Lincoln";
$name =~ s/(\w+) (\w+)/\2, $1/;# mixed $N and \N
print "$name\n";  # prints "Lincoln, Abraham"
print "$1\n";     # prints "Abraham"
```

Backreferences — Examples

1. Given an array — `@pal1 = (noon, naan, pip, pie, nine);`

   Create a regular expression that will identify the four character palindromes.

   Create a regular expression that will identify the three character palindromes.

2. Given an array — `@pal2 = ("wing on wing", "dollar for dollar","at the ball")`

   Create a regular expression that will identify the three word palindromes.

3. Given a string — `$string = "root  console  Mar 22 16:45"`

   Display this as: `Mar 22 16:45 ---> root on device console`

Modifying Strings with `tr///` or `y///`

- `tr///` and `y///` are identical commands, that transliterate (also called translate) specified characters in the topic into others
  `$topic =~ tr/`*`searchlist`*`/`*`replacementlist`*`/`
- No pattern is used, despite use of the binding operator
- The last character of *replacementlist* is replicated until *replacementlist* is the same length as *searchlist*
- The binding operator returns a count of characters replaced

```
$topic = "cats catch critters";
$res = $topic =~ y/cat/dog/; # $res is 10
print "$topic\n"; # prints dogs dogch driggers
```

- Different from global substitution:

```
$topic = "cats catch critters";
$res = $topic =~ s/cat/dog/g; # $res is 2
print "$topic\n"; # prints dogs dogch critters
```

Command Modifiers

- `m` and `s` patterns
  - `i` Ignore case.
  - `x` Ignore white space.
  - `s` Let the dot match a newline.
  - `m` Let anchors match a newline.
  - `o` Compile pattern only once.
- `m` only
  - `g` (list) find all matches
  - `g` (scalar) save position
  - `cg` Do not reset search position after a failed match.
- `s` only
  - `g` global replace
  - `e` evaluate right side
- `tr` and `y`
  - `c` Complement the search list.
  - `d` Delete specified characters.
  - `s` Squash duplicate characters.

Command Modifiers — Examples


```
1.    $string1 = "On day one we go to London";
2.    $string1 =~ s/on/ON/;
3.    $string1 =~ s/on/ON/g;
4.    $string1 =~ s/o n #comment/ON/xg;
5.    $string2 = "oooaa     eeee";
6.    $string2 =~ tr/oa e//s;
7.    $string3 = "dogs";
8.    $string3 =~ tr/dog/cat/;
9.    $string3 =~ tr/cs//d;
```

Module Subroutines

H4311S B.00
Module 8

Creating and Calling a Subroutine

```
sub mysub {
   my ($arg1, $arg2, @other_args) = @_;    # args
   my ($tmp, $retval, @atmp);      # "local" vars
      … # subroutine implementation code
   return $retval;     # return with an answer
}

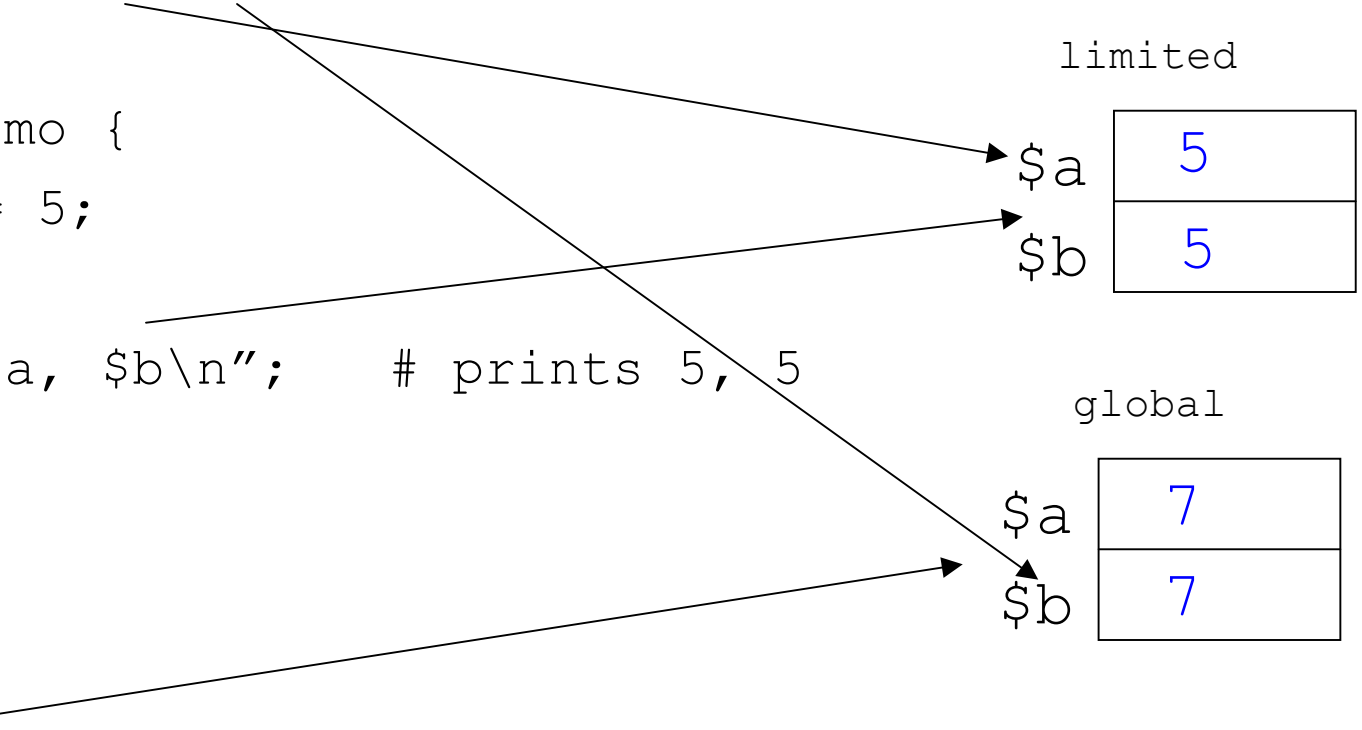
$result = mysub ( a, b, c, d, e );
$result = mysub a, b, c, d, e;
$result = &mysub(a, b, c, d, e);
```

Scope of Variables

- By default, variables in Perl have ***global*** scope
- The `my` and `local` list operators create variables of limited scope:
  - Variables “hide” previous variables with the same name
  - Variables may be initialized when created
  - Variables “disappear” when the current block completes
- The `my` list operator creates variables with ***static*** scope
  - Variables are accessible by code located within the current block
- The `local` list operator creates variables with ***dynamic*** scope
  - Variables are also accessible by any code called from within the current block

Example: Comparing `my` and `local`

```
sub print_ab {
  print "    $a, $b\n"; # prints 5, 7
}
sub scope_demo {
  local $a = 5;
  my $b = 5;
  print "  $a, $b\n";   # prints 5, 5
  print_ab;
}
$a = $b = 7;
scope_demo;
print "$a, $b\n";       # prints 7, 7
```



Subroutine Aliasing (Pass by Reference)

```
sub swap {
    my $tmp = $_[0];
    $_[0] = $_[1];
    $_[1] = $tmp;
}
    . . .
($a, $b) = (24, 7);
print "$a, $b\n";  # prints 24, 7
swap $a, $b;
print "$a, $b\n";  # prints 7, 24
```

Prototypes

- Prototypes may be used to specify the number and type of arguments a subroutine expects
- Prototypes are necessary when using *forward declarations*
- Use of prototypes is optional
- Example

```
sub mysub ($$@);        # forward declaration
   . . .
mysub 1, $i, @items;    # use
    . . .
sub mysub ($$@) {       # subroutine defined
    . . .
}
```

Preserving Arrays in a Subroutine Call

```
sub mysub {
    my ($aref1, $aref2, $v) = @_;
    my @a = @$aref1;
    my @b = @$aref2;
        . . .
}

mysub \@arr1, \@arr2, $var;
```

Special Variables

H4311S B.00
Module 9

Special Variables — Record Handling

$_	$ARG		Input value
$.	$INPUT_LINE_NUMBER	$NR	The line number in the current file handle. Reset by close.
$/	$INPUT_RECORD_SEPARATOR	$RS	Input record separator
$\	%OUTPUT_RECORD_SEPARATOR	$ORS	Output record separator
$,	$OUTPUT_FIELD_SEPARATOR	$OFS	Output field separator
$”	$LIST_SEPARATOR		Separator for the elements of a list
$;	$SUBSCRIPT_SEPARATOR	$SUBSEP	Default separator for simulated multi-dimensional arrays.

Special Variables — Formats

$%	$FORMAT_PAGE_NUMBER	Current page number in the output channel
$=	$FORMAT_LINES_PER_PAGE	Number of lines per output page
$-	$FORMAT_LINES_LEFT	Number of lines left on the current page
$~	$FORMAT_NAME	Name of current format
$^	$FORMAT_TOP_NAME	Top of page format (could be the header)

Special Variables — Regular Expressions

$n		The positional subexpression found in last match
$&	$MATCH	String matched by last pattern match
$`	$PREMATCH	The string preceding the last pattern matched
$’	$POSTMATCH	The string following the last pattern matched
$+	$LAST_PAREN_MATCH	The last match as a subexpression.

Special Variables — Process Information

$$	$PROCESS_ID	$PID	Process ID of the Perl program
$<	$REAL_USER_ID	$UID	UID of the process
$>	$EFFECTIVE_USER_ID	$EUID	Effective UID of the process
$(	%REAL_GROUP_ID	$GID	GID of the process
$)	$EFFECTIVE_GROUP_ID	$EGID	Effective GID of the process
$0	$PROGRAM_NAME		File name of the Perl script

Special Variables — Arrays and Hashes

@ARGV	Array of command line arguments passed to the script
@INC	Array of directories to search for scripts referenced by **`do`**, **`require`**, and **`use`**
%INC	Hash of file names included by **`do`** or **`require`** functions
%ENV	Hash of the current environment

Advanced Data Structures

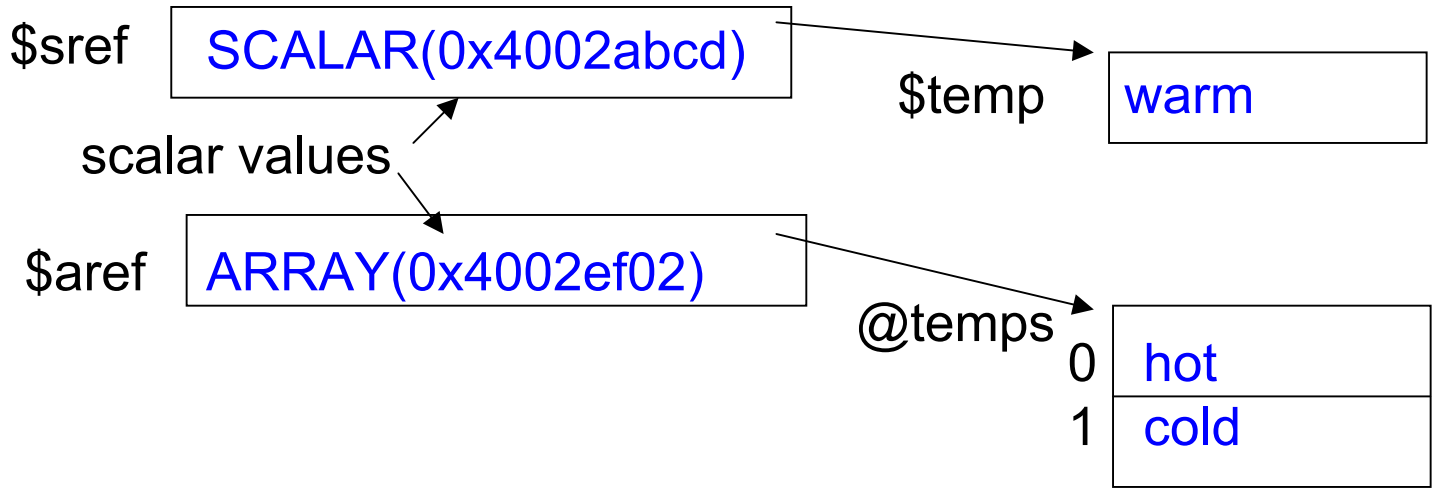
H4311S B.00
Module 10

What Is Possible

- records
  - simple
  - complex
- anonymous arrays and hashes
- multidimensional arrays
  - arrays of arrays
  - arrays of hashes
  - hashes of hashes
  - hashes of arrays
- linked lists

References

- Array and hash element values must be scalars
- References refer to a block of memory belonging to a scalar, array, or hash (or code)
- All references are scalars; what they refer to need not be



Creating References

- Use \ to create a reference to that variable's memory:

```
$sref = \$var;
$aref = \@arr;
$href = \%hsh;
```

- The value of the variable indicates the data type, and memory location:

```
print $sref; # prints "SCALAR(0x4001abcd)"
print $aref; # prints "ARRAY(0x400a0010)"
print $href; # prints "HASH(0x400e00aa)"
```

- Anonymous references can be created to arrays and hashes:

```
$anon_array = [value1, value2, value3];
$anon_hash = {key1, value1, key2, value2};
```

Using References

- SCALAR References

```
$var = "warm"; $sref = \$var;
print $sref;  print $$sref;
```

- ARRAY References

```
@temps = (hot, cold); $aref = \@temps;
print $aref;  print @$aref;
print $$aref[1];  print $aref->[1];
```

- HASH References

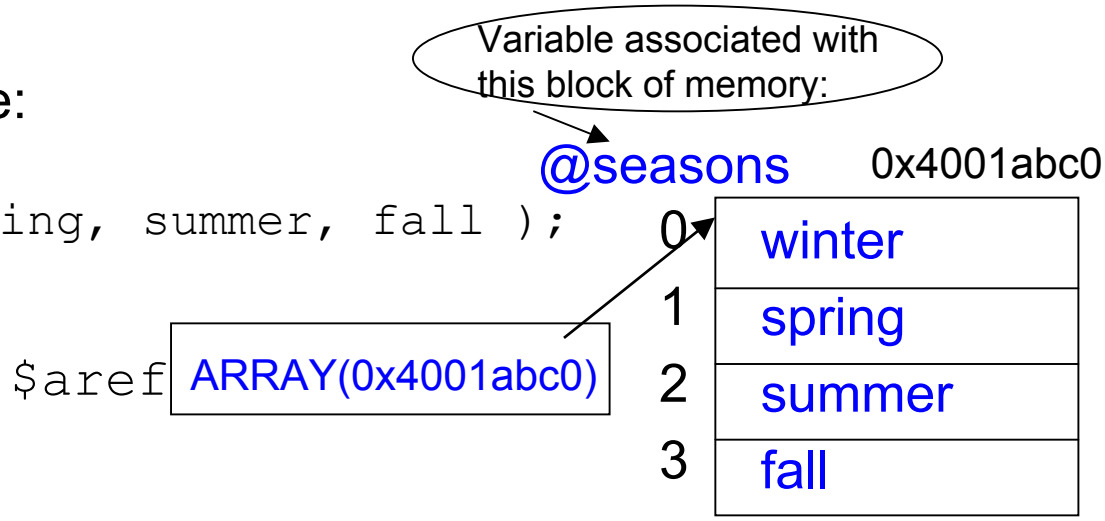
```
%book = (Title => "Lord of the Rings",
         Author => "JRR Tolkien");  $href \%book;
print $href;  print %$href;
print $$href{Title};   print $href->{Title};
```

Anonymous References

Reference to Named Variable:

```
@seasons = ( winter, spring, summer, fall );
$aref = \@seasons;

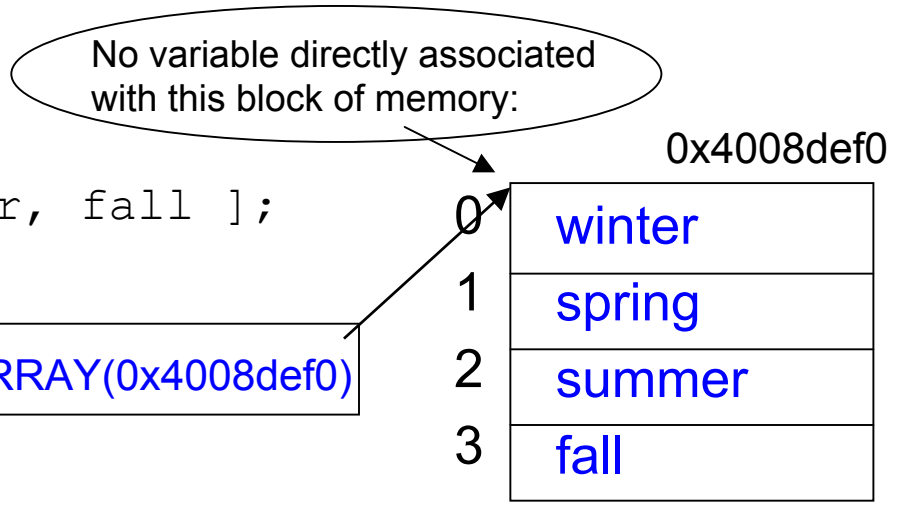
# print "summer":
print "$seasons[2]\n";
print "$$aref[2]\n";
print "$aref->[2]\n";
```



Anonymous Reference:

```
$aref = [ winter, spring, summer, fall ];

# print "summer":
print "$$aref[2]\n";
print "$aref->[2]\n";
```



Records

- A record is a list of related items.
- The items have a name and a value.
- Simple records are usually hashes, occasionally arrays, with scalar data values
- Complex records contain arrays and hashes
- A record is often implemented as an anonymous hash, using the hash constructor — `{...}`.
- Records are often stored in arrays or hashes, i.e. references to the records are stored.

Simple Record

- Hash implementation

```
%book = ( Title => "Lord of the Rings",
          Author => "JRR Tolkien" );
```

- Hash reference implementation

```
$book = { Title => "Lord of the Rings",
          Author => "JRR Tolkien" };
```

  - Access with `$$book{Title} or $book->{Title}`

- Array implementation

```
@book = ( "Lord of the Rings", "JRR Tolkien" );
```

- Array reference implementation

```
$book = [ "Lord of the Rings", "JRR Tolkien" ];
```

  - Access with `$$book[0] or $book->[0]`

Complex Records — Example

- Hash reference implementation

```
$boat = {
    "Manu"  => "Beneteau",
    "Model" => 311,
    "Year"  => 2000,
    "Color" => "white",
    "Features" => ["furling jib", "hot water"],
    "Options" => { "main" => "in mast furling",
                   "keel" => "bulb"}
};
```

- Access data using :
  - `$boat->{"Manu"}; $$boat{"Features"}[0];`
  - `$$boat{"Model"}; $$boat{"Options"}{"main"};`

Example: Array of Records

- `$library[1] = getbook();  #create a book`
- `print $library[1]{Title}; #print the title`

```
sub getbook{
   my ($title, $author);
   print "Enter a title: ";
   chomp ($title = <STDIN>);
   print "Enter the author: ";
   chomp ($author = <STDIN>);
   #return a reference
   return {Title=>$title, Author=>$author};
}
```

Arrays of Arrays

- Multidimensional arrays are created as arrays of references.

```
@array =  (      [one, two, three],
                 [dog, cat, bird],
                 [golden, tiger, canary]);
```

- `$array[0]` is (one, two, three)
- `$array[1]` is (dog, cat, bird)
- `$array[2]` is (golden, tiger, canary)
- `$array[1][2]` is bird
- This could also be done using an anonymous array constructor instead of a list.

Arrays of Hashes

- `@dogs = (`

```
        { “dog” => “lab”,
          “name” => “rover” ,
          “size” => “big” },
        { “dog” => “spaniel”,
          “name” => “bowser” ,
          “size” => “medium” }  )
```

- `$dogs[0]{“dog”}` refers to `lab.`

Hashes of Hashes

- `%pets =(dogs => {mine => obedient,`
  `yours => untrained},`
  `cats => {mine => independent,`
  `yours => undisciplined},`
  `hamsters=> {mine = >perfect,`
  `yours => unmotivated});`
- `$pets{cats}{mine}` refers to `independent.`

Hashes of Arrays

- `%animals = (`
  `dogs => [spaniel, poodle, lab],`
  `cats => [persian, tabby],`
  `birds => [canary, duck, goose, turkey] );`

- `$animals{dogs}[1]` is `poodle`
- `$animals{cats}[0]` is `persian`

Linked List

```
sub make_node{
    print "Enter  record: ";
    chomp ($value = <STDIN>);
    my $node = {"value" => $value, "next" => $next};
    return  $node;
}
   . . .
if (defined $head){
    $last_node = find_last_node($head); #see notes
    $last_node{next} = make_node();
}
else{
    $head = make_node();
}
```

The CGI Protocol

H4311S B.00
Module 11

The CGI Protocol Defined

- Common Gateway Interface is
  - a **protocol**, not a programming language
- Can be implemented using any language
  - UNIX shells, C, C++, Visual Basic, Java, but especially Perl
- Works cross-platform
  - UNIX, Linux, NT
- A **protocol** is an accepted method of doing something
  - a set of conventions governing the treatment and especially the formatting of data

CGI's Role

- CGI is the glue that holds the web together.
  - Typically sandwiched between HTML forms
- A client completes a form to provide needed information to the program running on the server.
- The CGI script is executed on the server in real time.
- Results are relayed back to the client.
- A cheap disclaimer….
  - We will keep HTML as simple as possible.
- The module, `cgi.pm` will be deferred until the next unit in this course.
  - This let's us get a better look at the data flow.

Creating a Form

```
print  "Content-Type:  text/html\n\n";
print `
<FORM ACTION="http://www.servername.com/cgi-
bin/task.cgi"
       METHOD=POST>
<B>Select task:</B>
<SELECT NAME="task">
<OPTION VALUE="check_daemons">check daemons
<OPTION VALUE="kill_old_users">kill old users
</SELECT>
<INPUT TYPE="submit" VALUE="submit task">
`;
```

Text Area and Radio Buttons on a Form

```
print “Content-Type:  text/html\n\n”;
print ‘<FORM ACTION=“http://www.servername.com/cgi-
bin/task.cgi” METHOD=GET>
First Name: <INPUT TYPE="TEXT" NAME="firstname"
SIZE="25"><BR>
Last Name : <INPUT TYPE="TEXT" NAME="lastname"
SIZE="25"><BR>
<INPUT TYPE=“radio” NAME=“job_title” VALUE=“S”>Sysadmin
<INPUT TYPE=“radio” NAME=“job_title” VALUE=“N”>Netadmin
<INPUT TYPE=“radio” NAME=“job_title”
VALUE=“W”>Webmaster
<INPUT TYPE=“submit” VALUE=“submit”>
’;
```

Security

- Security is naturally a concern.
- The ISP or webmaster will determine *if* and *where* CGI scripts will be allowed to run.
- Three levels:
  - `/opt/apache/cgi-bin` (more secure)
  - allow users to maintain their own directory for CGI scripts (less secure)
  - any directory, – the program name must end in `.cgi` (insecure)
- If user's are allowed to maintain their own CGI scripts a configuration change will be made to allow `public_html`
  - this path is appended to `~user`.

    For example, the script called by

    `http://r208w100/~instr/prog.cgi`

    will be `/home/instr/public_html/prog.cgi`

The Issue of Pathnames

- Path names are not the same as URLs.
  - Structurally they *look* similar.
  - URLs may have path names embedded, which makes them look like path names.
- UNIX path names (either absolute or relative) are literal.
  - You know your starting place (you can see it).
- CGI pathnames are composites.
  - They have “roots” defined by the webmaster in configuration files.
  - Check `httpd.conf` in `/opt/apache/conf/httpd.conf`
  - Look for `DocumentRoot`, `UserDir`, `ScriptAlias`, `ServerRoot`.
- Test to verify your discoveries.

CGI Programs as a Security Issue

- CGI programmers also have security responsibilities.
- CGI programs are tempting targets.
- Adopt a defensive mindset.
  - remember, just because you're paranoid doesn't mean someone isn't really out to get you.
- Identify lines in your code that grant access to the server.
- Scrutinize them and test rigorously for
  - valid (expected) data, or ranges
  - origin (Is this data provided from where I expect?)
  - Path names require extra vigilance. Don't allow double dots (`..`) as this could be an attempt to get to `../../etc/passwd` or the like.

Environmental Variables in Programs

- Variables live in `%ENV` hash

`DOCUMENT_ROOT`	— Absolute path of the server's root directory
`GATEWAY_INTERFACE`	— The version of CGI the server's running
`HTTP_ACCEPT`	— A list of supported MIME types
`HTTP_USER_AGENT`	— Name/version of browser
`QUERY_STRING`	— String resulting from form data
`REMOTE_ADDRESS`	— IP address of user's system
`REMOTE_HOST`	— Host name of user's system
`REQUEST_METHOD`	— `METHOD` of `HTML` form (`GET` or `POST`)
`SCRIPT_NAME`	— Current program's relative pathname
`SERVER_ADMIN`	— Email address of server administrator
`SERVER_NAME`	— Domain name or IP address of server
`SERVER_PORT`	— Port the request was sent to (80 default)
`SERVER_PROTOCOL`	— Name and version of request protocol
`SERVER_SOFTWARE`	— Name and version of server software

Debugging CGI Scripts

- Debugging Perl scripts is easy — use `perl -w` or `perl -c` or `perl -d`.
- Debugging CGI scripts is difficult.
  - distractions that are side effects of the client-server architecture (name lookups, connectivity issues, cross-platform issues, etc)
- Development environment for testing
  - ideally under your control
- Some things to look for:
  - The HTTP header line `(print "Content-Type: text/html\n\n";)`
  - Try running the script with `perl -c` before browser invocation.
  - To let you see what happens *before* the "500 Server Error" add:

```
#!/usr/bin/perl
   $| = 1;
   print "Content-Type: text/plain\n\n";
```

Perl Modules

H4311S B.00
Module 12

What Is a Module?

A module is

- Perl script that another programmer wants to share
- located at CPAN web sites
- a combination of C source and header files, configuration files, documentation, and scripts
- accessed as a zipped tar file
- available for web, networking, windows, X11, etc.
- can be improved on and resubmitted

Building and Installing Modules

- Go to the web site and copy the file to the local server.
- DECOMPRESS
  - Use the proper unzip utility to restore the tar file.
- UNPACK
  - Untar the file.
- BUILD
  - Make the unpacked `module` directory your current directory.
  - Execute `perl Makefile.PL`
  - Execute `make`
  - Execute `make test`
- INSTALL
  - Execute `make install`

Sockets

- A socket can be a port at an IP address that receives data.
- A socket can be a port at which a local application receives data.
- A server “listens” at a port.
- A client is a program that sends information to or requests information from a server at a specific port.
- There are two different types of messages, streams and datagrams.

Sockets — Example

SERVER

```
1.  Use IO::Socket;
2.  $sock=IO::Socket::INET ->new
3.  (LocalPort => 12345,
4.  Type => SOCK_STREAM,
5.  Reuse => 1,
6.  Listen = 5) or die “message”;
7.  while ($client = $sock->accept){
8.  $line = <$client>;
9.  print $line;
    }
10. close ($sock);
```

CLIENT

```
1.  Use IO::Socket;
2.  $sock=IO::Socket::INET ->new
3.  (PeerAddr => ‘hostname’,
4.  PeerPort => 12345,
5.  ype => SOCK_STREAM,
6.  Proto => ‘tcp’,) or die “message”;
7.  while (more_to_send){
8.  $line = data_to_send;
9.  print $sock $line;
    }
10. close ($sock);
```

CGI

- The CGI module is a standard module.
- The CGI module generates the web pages dynamically.
- `<STDIN>` and `<STDOUT>` now use the web browser.
- The screen is created by printing the HTML commands to the browser.
- The CGI methods produce HTML code dynamically.

CGI — Example

```
use CGI;
$page = new CGI;
print   $page -> header(),
    - $page -> start_html(),
    - $page center($page -> h1("Hello World")),
    - $page start_form(),
    - $page -> textarea(
          -name => 'My Text Area',
          -rows => 10,
          -columns => 40),
    - $page -> end_form(),
    - $page -> end_html();
```