Welcome to my CPS 600 Tutorial Project

-- PERL

You are the visitor number 007280 since February 1, 1996 to look at my Information Age project page at 12:25 PM EST.

Table of Contents

How to read this tutorial Perl project?

- Chap 1 introduces Perl briefly. It includes "What is Perl?", "Who creates Perl?", "Perl’s license" and so on.
- Chap 2 ~ Chap 11 introduce basic ideas of Perl. These chapters are based on Perl4. However, you will not feel much different when you use these basic ideas on Perl5. Please check the Chap 14 section for the comparison between Perl4 and Perl5. You can refer to the ***WWW sites for Perl*** and/or ***References*** for advanced Perl information.
- Chap 12 includes links to on line man pages and manuals.
- Chap 13 includes examples/answers for 2 famous books -- the ***Programming Perl (Camel book)*** and the ***Learning Perl (Llama book)***.
- Chap 14 compares the difference between Perl4 and Perl5 briefly. Generaly speaking, Perl5 adds more features but the compatibility is very high. You can check ***Metronet Perl5 Info*** in the ***WWW sites for Perl*** for more Perl5 new features.
- Chap 15 includes some WWW Perl sites for you to refer. They may have latest updated information of Perl. You can also download Perl from the FTP Archive listed in this chapter.

---

1. Introduction to Perl

---

1. What is Perl?

Perl is short for "*__P__ractical __E__xtraction and __R__eport __L__anguage*". Perl integrates the best features of **Shell programming, C,** and the UNIX utilities -- **grep, sed, awk** and **sh**.

2. Who creates Perl?

The inventor of Perl is **Larry Wall**.

3. Perl’s license

Perl is distributed under the **GNU Public License**. That means it is essentially **FREE**.

4. Which platforms can Perl run on?

It can be run on various platforms such as UNIX, UNIX-like, Amiga, Macintosh, VMS, OS/2, even MS-DOS and maybe more in the near future.

5. Where can I get Perl's information?

You can get any information from the USENET newsgroup "**comp.lang.perl**". There is information for obtaining Perl, solving the problems, ..., etc.. There are also many experts monitoring this news group including the inventor -- Larry Wall. That means you might get response in a minute.

6. How can I begin a perl program?

The simplest way to do it is include the following line at the beginning of your perl file:

```
    #!/bin/perl (or the path Perl located on your system)

Some may have:

    #!/usr/local/bin/perl

    your program
         :
         :
         :
```

With this line, the shell knows where to look for Perl to run the program.

7. Is it a Perl program or a Perl script?

It's up to you. If you prefer to call it Perl program, then call it program. If you like script more, then call it script.

8. Is Perl difficult to learn?

No, it's not. Many people(including myself) think Perl is easy to learn. There are some reasons:

- Since most of Perl is derived from some tools, utilities, programming languages that you may be familiar with. For example, you will think it is pretty easy for you if your are familiar with C, shell, awk or sed.
- You do not need to remember many things to be able to use Perl. For example, you can start your Perl script(program) by the following:

  ```
  #!/bin/perl
  print "Say Hi to Neon!";
  ```

- You can get your result right away. You don't have to compile your programs every time after you change something.

9. Should I program everything in Perl?

As a matter of fact, you can do anything in Perl. You, however, should not. Why? You should use the most appropriate tool for your job. Some people use Perl for shell programming, some people use Perl to replace some C programs. Perl, however, is not a good choice fot very complex data structures.

---

2. Scalar Data

A *scalar* is the simplest kind of data that perl manipulates. A scalar can be either a number or a string of characters.

Numbers

Even though we can specify integers, floating-poit number, ..., etc.. However, internally, Perl computes only with double-precision floating-point values.

Strings

- Single-Quoted Strings
  A single-quoted string is a sequence of characters enclosed in single quotes.
  One thing to notice, some special characters like newline will not be interpreted within a single-quoted string. For example:

```
’hello’      # 5 characters
’don\’t’     # 5 characters
’hello\n’    # 7 characters
```

- Double-Quoted Strings
  As to a double-quoted string, it is much like a C string. The *Backslash Escapes* work within the double-quoted strings. The complete list of double-quoted string escapes is listed below:

Escape	Meaning
\n	Newline
\r	Return
\t	Tab
\f	Formfeed
\b	Backspace
\v	Verticle tab
\a	Bell
\e	Escape
\cC	CTRL + C
\\	Backslash
\"	Double quote
\l	Lowercase next letter
\L	Lowercase all following letters until \E
\u	Uppercase next letter
\U	Uppercase all following letters until \E
\E	Terminate \L or \U

Another feature of double-quoted strings is that they are *variable interpolated*. That means that some variable names within the string are replaced by their current values when the strings are used.

Operators

An operator generates a new value from one or more values. In Perl, the operators and expressions are generally a superset of most programming languages such as C. One thing you might be interesting is the operators between Numbers and Stings are different. The comparison is listed below:

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

Let’s talk more about the scalar data from now on.

○ Scalar Variables

A variable is a name for a container that holds one or more values. A *scalar variable* holds a single scalar value. Scalar variable names lead by a dollar sign($) followed by a letter and then possibly more letters, digits or underscores. Be careful, uppercase and lowercase letters are distinct. That means $H and $h are different variables.

○ Operators on Scalar Variables

■ Assignment operator

The operator we use most commonly is the *assignment*. The examples are listed below:

```
$a = 5;                 # assign 5 to $a
$b = 4;                 # assign 4 to $b
$c = $a * $b;           # assign 20 to $c
$d = "Hello, World";    # assign a string to $d
```

■ Binary Assignment Operator

Like C, Perl has a shorthand for the operation of altering a variable -- the *binary assignment operator*. The following expressions are equivalent:

```
$a = $a + 2;    # Without the Binary Assignment Operator
$a += 2;        # With the Binary Assignment Operator
```

■ Autoincrement(++) and Autodecrement(--)

Like C, Perl provides the *Autoincrement* and *Autodecrement* to simplify the expressions. The following expressions are equivalent:

```
$a +=1; # With the Binary Assignment Operator
$a++;   # With Postfix Autoincrement
$++a;   # With Prefix Autoincrement

$b = 3;
$c = $b++;      # $b is 4 and $c is 3 after this expression
$d = ++$b;      # both $b and $d are 5 after this expression
```

Same to Autodecrement.

- **chop() Operator**

This operator removes the last character from the string variable.

```
$s = "Hello";
chop($s);       # $s becomes "Hell"
```

It is useful when you read a value from < STDIN >. You can use chop() operator to remove the newline character which may cause problems in the future.

- **Interpolation of Scalar into Strings**

Let the following examples explain what it means:

```
$a = "zoo";
$b = "elephants";
$c = "We can see $b in the $a.";
# $c is "We can see elephants in the zoo." now
```

- **print() Operator**

Whenever you want to print something on screen, you can just use the print() operator.

---

3. Array and List Data and Associative Arrays

An *array* is an ordered list of scalar data. Each element of the array is a separate scalar variable with the corresponding value.

Array Variables

Array variables hold a single array value(0 or more scalar value). The array variable names are similar to scalar variable names except the leading character. The scalar variable name begins with a dollar sign($) but array variable names are:

- List Array
  It begins with an at sign(@).
- Associative Array
  It begins with a percent sign(%).

Operators for a List Array:

- **Assignment(=)**

  The assignment operator gives an array variable a value. It is an equal sign(=) like scalar assignment operator. Perl decides whether the assignment operator is scalar or array assignment according to the variables it will assign to.
  Here are some examples:

```
@test1 = ("Hello", "World");# It has 2 elements now.
@test2 = @test1;            # test2 is the same as test1.
@test3 = (@test1,"pal");    # test3 is ("Hello","World","pal")

($t1, $t2) = (6, 12);       # assign 6 to $t1, 12 to $t2
($t1, $t2) = ($t2, $t1);    # swap $t1 and $t2
@test = (1, 2, 3);
@test = (0, @test);         # @test is now (0, 1, 2, 3)
@test = (@test, 4);         # @test is now (0, 1, 2, 3, 4)
($tmp, @test) = @test;      # @test is now (1, 2, 3, 4), $tmp is 0

$length = @test;            # $length is 4 now
($length) = @test;          # $length is 1 now(the first element of
                              @test)
```

- **Element Access**

  We deal with the whole array by the assignment operator so far. What if we want to access some specific elements of the array? To do this, we need to use subscripting operator to refer an array element by an index. The number begins at 0 and increases 1 for each element. Here are some examples:

```
@test = (1, 2, 3);
$t1 = @test[2];             # $t1 is 3 now
$test[1] = 6;               # @test is (1, 6, 3) now
$test[0]++;                 # @test is (2, 6, 3) now
$test[0] *= 5;              # @test is (10, 6, 3) now
```

- **push() and pop()**

  These 2 operations act like a stack operation. push() will insert an element at the end of the array which is being operated. pop() will remove the last element from the array. Here are some examples:

```
@test = (1, 3, 5);
push(@test, 2, 4, 6);       # @test is (1, 3, 5, 2, 4, 6) now
$last = pop(@test);         # @test is (1, 3, 5, 2, 4) and $last
                              is 6 now
```

Both push() and pop() will take an array variable name as the first argument.

- **shift() and unshift()**

  Unlike push() and pop() do things at the end of the array, unshift() and shift() do things at the beginning of the array. Here are some examples:

  ```
  unshift(@test, 7, 8);       # @test is (7, 8, 1, 3, 5, 2, 4)
  $first = shift(@test);      # @test is (8, 1, 3, 5, 2, 4) and
                                $first is 7
  ```

- **reverse()**

  This operator reverses the order of the element if the array. For example:

  ```
  @test1 = (1, 2, 3);
  @test2 = reverse(@test1);   # @btest2 is (3, 2, 1)
  ```

  However, @test1 is unchanged. reverse() works on a copy not the original one. If you want to reverse the array, do the following:

  ```
  @test1 = reverse(@test1);
  ```

- **sort()**

  It sorts the elements of the array as single strings in ascending ASCII order without changing the original list. For example:

  ```
  @test1 = (10, 5, 3, 7, 47, 8);
  @test2 = sort(@test1);      # @test2 is (10, 3, 47, 5, 7, 8)
  ```

  Again, if you want to change the original list, do the following:

  ```
  @test1 = sort(@test1);
  ```

Associative Arrays

An associative array is just like a list array. The difference between an associative array and a list array is that the list array uses non-negative integers as index values but the associative array uses **arbitrary** scalars. These scalars, also called *keys*, are used to retrieve the corresponding values from the associative array.

One thing we have to notice is that there is no particular order for the elements of an associative array. Whenever we want to find some specific values, we use the keys to find them. We do not have to worry how we can find them because Perl has the internal order to do this.

Most of time, people want to access the elements of the associative array rather the entire array. At this time, we need the keys to do this. The associative array is represented as:

```
    %test
```

and an element of an associative array is represented as:

```
    $test{$key}                  # Notice! The leading character is a
                                   dollar sign($) and so is the key.
```

How do we create and/or update an associative array? Here are some examples:

```
    $test{1} = "Hello";          # creates key 1 and value "Hello"
    $test{2} = 100;              # creates key 2 and value 100
```

We can also assign the key-value pairs to a list array.

```
    @test1 = %test;              # @test1 is either (1, "Hello", 2, 100)
                                   or (2, 100, 1, "Hello")
```

The order of the key-value pair is arbitrary and cannot be controlled. Perl has its own logic to have more efficient access. Of course, the list array can copy its values to an associative array.

```
    %test2 = @test1;             # %test2 is just like %test now.
```

Operators for Associative Arrays:

- **keys()**

  This operator can be used while we want to access a list of keys of the associative array. As a matter of fact, it returns the odd-number (1, 3, 5, 7, 9, ...) elements of the array.

  ```
      @test3 = keys(%test);       # @test3 is either (1, 2) or (2, 1)
  ```

- **values()**

  Instead of returning the keys of the associative array, this operator returns values of the associative array. That is, it returns the even-number (2, 4, 6, 8, ...) elements of the array.

  ```
      @test4 = values(%test);     # @test4 is either ("Hello", 100) or
                                    (100,  "Hello")
  ```

- **each()**

  You can, of course, access both keys and values of the associative array by using each() operator.

  ```
      $personinfo{"011-88-6257"} = "John";
      $personinfo{"323-56-2943"} = "Tom";
      $personinfo{"242-54-2489"} = "Sam";
      print "SSN \t\t NAME\n";
      print "---------------------\n";
      while ( ($ssn, $name) = each(%personinfo) )
      {
          print "$ssn \t $name\n";
      }
  ```

```
The result looks like:

SSN              NAME
----------------------
242-54-2489      Sam
011-88-6257      John
323-56-2943      Tom
```

As mentioned above, the order of key-value pair is arbitrary. In this case, we can see the result. We created the associative array by "John --> Tom --> Sam" but got "Sam --> John --> Tom".

We borrowed the **while()** statement that we will discuss later on in the section of control structures.

- **delete**

  What if we want to remove elements from an array? No problem. Perl provides the delete operator to do this. However, it will remove both key and value from the associative array. We apply the previous example to explain. If we add the following lines into the previous program, we will get the following result:

```
print %personinfo,"\n";
delete $personinfo{"011-88-6257"};
print %personinfo,"\n";

The result looks like:

SSN              NAME
----------------------
242-54-2489      Sam
011-88-6257      John
323-56-2943      Tom
242-54-2489Sam011-88-6257John323-56-2943Tom
242-54-2489Sam323-56-2943Tom
```

4. Control Structures

There are several statements provided by Perl.

- ## if/unless statement

  This is like other structured programming languages.

```
if (expression) {
   statement 1;
   statement 2;
       :
       :
} else {
   statement a;
   statement b;
       :
       :
}
```

One thing we have to know is that the control expression is evaluated for a **string** value. That means, there will be no change if it is already a string but it will be converted to a string if it is a number.

On the other hand, if you want to do the reverse way, you can apply **unless** instead of if statement. The meaning of unless statement is "If the expression is not true, then ...".

You can also apply **elsif** as many times as you wish to expand your if/else statement.

```
if (expression 1) {
       :
       :
} elsif (expression 2) {
       :
       :
} else {
       :
       :
}
```

○ while/until statement

We have seen how to use while statement in the previous section. We, now, formally introducw its syntax:

```
while (expression) {
   statement 1;
   statement 2;
      :
      :
}
```

We will execute the statements inside the brackets if the expression is true via while statement. However, we may want to do the other way. Instead of true value of the expression, we can let **until** statement test the false value and do the execution.

```
until (expression) {
   statement 1;
   statement 2;
      :
      :
}
```

○ for statement

This statement is pretty much like C’s for statement. Here is the syntax:

```
for (initial_exp; test_exp; increment_exp) {
       statement 1;
       statement 2;
          :
          :
    }
```

- **foreach statement**

  This statement is pretty much like the C-shell’s foreach. It takes a list of values and assigns them one at a time to a scalar variable and execute a block of statement. The syntac is:

```
    foreach $s (@list) {
       statement 1;
       statement 2;
          :
          :
    }
```

---

5. Basic I/O

- **Input from < STDIN >**

  The usage is:

```
    $test1 = < STDIN >;
    @test2 = < STDIN >;
```

  The first one reads a value from the standard input terminated with a newline character. The second one reads as many lines as you want to until you press "CTRL+D" to terminate it.

- **Diamon Operator (<>)**

  Perl also provides another way to read input via the diamond operator(<>). The difference between the previous one and this one is it reads data from file or files which are specified on the command line. However, if you don’t specify any filename on the command line, it will read data from standard input.

- **Output to STDOUT**

  Perl uses two operators to write to standard output:

  - **print**

    This is the normal output operator. The usage is like:

```
    print "Hello, World.\n";
    print 123+456;
```

- **printf**

  This is the formatted output. Again, it is pretty much like C's printf operator. The usage is like:

  ```
  $name = "John";
  $age = 23;
  $ssn = "121-66-3214";
  printf "My name is %10s.  I am %3d years old.  My SSN is %11s.", $

  The result looks like:

  My name is John.  I am 23 years old.  My SSN is 121-66-3214.
  ```

---

6. Formats

Perl also provides the notion of a simple report template which is called **format**. A format in Perl contains two parts: constant part (the column header, labels, fixed text, ...) and variable aprt (current data).

Using a foramt consists of doing three things:

1. Defining a format.
2. Loading up the data to be printed into the variable portions of the format (fields).
3. Invoking the format.

Usually, the first step is done once and the other two are done repeatedly.

- ## Defining a format

  You need to use a format definition if you want to have a format. The format definition can be anywhere in the program text. A format definition looks like:

  ```
  format formatname =
  fieldline
  value_1, value_2, value_3
  fieldline
  value_4, value_5
  .
  ```

  Here comes an example for an address label:

  ```
  format ADDRESSLABEL =
  =============================
  @<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  $name
  @<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  ```

```
$address
@<<<<<<<<<<<<<, @< @<<<<<<<<<<<
$city, $state, $zipcode
=============================
.
```

○ Invoking a format

You may want to invoke the format definition in your Perl program. You can do it by using the **write** operator.

```
$name = "John Starks";
$address = "1234 Erie Blvd.";
$city = "Syracuse";
$state = "NY";
$zipcode = "13205";
write ADDRESSLABEL;

The result looks like:

=============================
John Starks
1234 Erie Blvd.
Syracuse, NY 13205
=============================
```

○ Fieldholders

■ Text Fields

You can get left-justified field by using an at sign and followed by left angle brackets(@<<<<). Does it matter with the number of the left angle brackets? Yes, it does. You can hold as many as characters as you specify the number of the left angle brackets plus one. Why? Since the at sign(@) counts.

Similarly, we can have right-justified field by @>>>> and centered field by @||||.

■ Numeric Fields

There is also a fieldholder for numbers. Instead of (<), (>) or (|), the at sign(@) is followed by hash sign(#). The format definition may look like:

```
format ACCOUNT =
Deposit: @#####.## Withdraw: @#####.## Balance: @#####.##
$deposit, $withdraw, $deposit-$withdraw
.
```

■ Multiline Fields

Perl also provides multiline fields to process more than one line. The fieldholder is denoted by @*. Here is the example:

```
format TEST =
=====================
@*
$test
=====================
.


$test = "test1\ntest2\ntest3\ntest4\n";
write;

Then, we will get the result:

=====================
test1
test2
test3
test4
=====================
```

- ### The Top-of-Page Format

  Perl also allows you to have your own top-of-page format definition. It may look like this:

  ```
  format ADDRESSLABEL_TOP =
  Address label page @<
  $%
  .
  ```

  The ($%) is used to display the page number. As a matter of fact, this variable is the number of times the top-of-page format has been called for a particular file handle.

---

7. Regular Expressions

Regular Expressions

A *regular expression* is a pattern, a template to be matched against a string. Regular expressions are used frequently by many UNIX programs, such as *awk, ed, emacs, grep, sed, vi* and other shells. Perl is a semantic superset of all of these tools. Any regular expression that can be described in one of the UNIX tools can also be written in Perl, but not necessarily using exactly the same characters.

In Perl, we can speak of the string test as a regular expression by enclosing the string in slashes.

```
while (<>) {
   if (/test/) {
      print "$_";
   }
}
```

What if we are not sure how many e’s between "t" and "s"? We can do the following:

```
while (<>) {
   if (/te*st/) {
      print "$_";
   }
}
```

This means "t" is followed by zero or more e's and then followed by "s" and "t".

We, now, introduce a simple regular expression operator -- **substitute**. It replaces the part of a string that matches the regular expression with another string. It looks like the **s** command in *sed*, consisting the letter **s**, a slash, a regular expression, a slash, a replacement string and a final slash, looks like:

```
s /te*st/result/;
```

Here, again, the $_ variable is compared with the regular expression. If they are matched, the part of the string is replaced by the replacement string ("result"). Otherwise, nothing happens.

Pattern

A regular expression is a pattern.

- **Single-Character Patterns**

  The simplest and most common pattern-matching character in regular expression is a single character that matches itself. Another common pattern matching character is the dot ".". It matches any single character except the newline character(\n).

  A pattern-matching **character class** is represented by a pair of open and close square brackets, and a list of characters inside. If you want to put some special characters like], -, ..., you need to use backslash(\). There is a shorter expression of a long and consecutive list of numbers or characters. Use a dash(-) to represent.

  ```
  [0123456789]        # all digits
  [0-9]               # the same with above
  [0-9\]]             # all digits and right square bracket
  [a-z0-9]            # all lowercase letters and digits
  [a-zA-Z0-9_]        # all letters and digits and underscore
  ```

  There is another ***negated*** *character class* which is reverse to the character class. It leads by a caret(^) character right after the left square bracket. This negated character class matches any single character that is **not** in the list. For example:

  ```
  [^0-9]              # match any single non-digit
  [^aeiouAEIOU]       # match any single non-vowel
  [^\]]               # match any character except a right square bracket
  ```

  Some readers might think, it is bothersome to type so many characters everytime. Is there an abbreviation for digits and/or characters? The answer is "Yes". Perl provides some predefined character classes for your convenience.

  ```
  Construct  | Equivalent Class | Negated Construct | Equivalent Negated Clas
  ```

```
----------+-----------------+-----------------+----------------------
\d (digits)| [0-9]          | \D (non-digits)  | [^0-9]
\w (words) | [a-zA-Z0-9_]    | \W (non-words)   | [^a-zA-Z0-9_]
\s (space) | [\f\n\r\t]      | \S (non-space)   | [^\f\n\r\t]
```

- **Grouping Patterns**

As we saw before, the asterisk(*) can be used in grouping pattern. It means "zero or more" of the character it follows. We, now, introduce other two grouping patterns. The first one is the plus sign(+), meaning "one or more" of the character it follows. The second one is the question mark(?), meaning "zero or one"of the character it follows.

We may, sometimes, need to specify the number of characters we want to handle. We, therefore, need the concept of general multiplier. The general multiplier consists of a pair of matching curly braces with one or two numbers inside. The format will look like:

```
    /a{3,8}/            # must be found 3 a’s to 8 a’s
    /a{3,}/             # means 3 or more a’s
    /a{3}/              # exactly 3 a’s
    /x{0,3}/            # 3 or less a’s
```

Now, let’s think about those 3 grouping patterns mentioned before. "*" is jukst like {0,}, "+" is just like {1,} and "?" is like {0,1}.

Another grouping pattern operator is a pair of open and close parentheses around any part pattern.

```
    /a(.)b\1/;          # It can be matched by "axbx"
```

What if there are more than one pair of parentheses in the regular expression? If this is the case, the second pair of parentheses is referenced as \2, and so on.

```
   /a(.)b(.)c\1d\2/;    # It can be matched by "axbycxdy"
```

Another usage is in the replacement string of a substitute command.

```
    $_ = "a xxx b yyy c zzz d";
    s/b(.*)c/s\1t/;     # $_ becomes "a xxx s yyy t zzz d".
```

Another grouping pattern is **alternation** in form of a | b | c. It can also be used for multiple characters. As a matter of fact, it is better to use a character class for single character alternatives.

```
    /sony|panasonic/;   # match either sony or panasonic
```

Here are some more examples of regular expressions, and the effect of parentheses:

```
    abc*                # abc, abcc, abccc and so on
    (abc)*              # abc, abcabc, abcabcabc and so so on
    ^a|b                # a at the beginning of a line or b anywhere
    ^(a|b)              # either a or b at the beginning of a line
    (a|b)(c|d)          # ac, ad, bc, or bd
    (red|blue)pen       # redpen or bluepen
```

Selecting a Different Target

Sometimes, we do not want to match patterns with the $_ variable. Perl provides the =~ operator to help us for this problem.

```
$test = "Good morning!";
$test =~ /o*/;              # true
$test =~ /^Go+/;            # also true
```

One thing we have to notice again here is we never store the input into a variable. That means, if you want to match this input again, you won’t be able to do so. However, this happens often.

Ignoring Case

We, sometimes, may want to consider patterns with both uppercase and lowercase. As we know, some versions of *grep* provides -i flag indicating "ignore case". Of course, Perl has a similar option. You can indicate the ignore case option by appending a lowercase "i" to the closing slash, such as /patterns/i.

```
< STDIN > =~ /^y/i;         # accepts both "Y" and "y"
```

Using a Different Delimiter

We may, sometimes, meet such a situation:

```
$tmp =~ /\/etc\/fstab/;
```

As we know, if we want to include slash characters in the regular expression, we need to use a backslash in front of each slash character. It looks funny and unclear. Perl allows you to specify a different delimiter character. Precede any nonalphanumeric character with an "m".

```
m@/etc/fstab@               # using @ as a delimiter
m#/etc/fstab#               # using # as a delimiter
```

Special Read-Only Variables

There are three special read-only variables: 1. $&, which is the part of the string that matched the regular expression. 2. $‘, which is the part of the string before the part that matched. 3. $’, which is the part of the string after the part that matched. For example:

```
$_ = "God bless you.";
/bless/;
# $‘ is God " now
# $& is "bless" now
# $’ is " you." now
```

Substitutions

We have know the simple form of the substitution operator: s/old_regular_expr/replacement_string/. We now introduce something different. If you want to replace all possible matches instead of just the first match, you can append a **g** to the closing slash.

```
$_ = "feet feel sleep";
s/ee/oo/g;                  # $_ becomes "foot fool sloop"
```

You can also use a scalar variable as a replacement string.

```
$_ = "Say Hi to Neon!";
$new = "Hello";
s/Hi/$new/;                 # $_ becomes "Say Hello to Neon!"
```

You can also specify an alternate target with the =~ operator.

```
$test = "This is a book.";
$test =~ s/book/desk/;      # $test becomes "This is a desk."
```

The split() and join() operators

In Perl, there are two operators used to break and combine regularu expressions. They are **split()** and **join()** operators.

- **The split() operator**

  The split() operator takes a regular expression and a string and looks for all occurrences of the regular expression within the string.

  ```
  $test1 = "This is a project for CPS600.";
  @test2 = split(/\s+/,$test1);
  # split $test1 by using " " as delimiter
  # @test2 is ("This","is","a","project","for","CPS600.")
  ```

  If we change $test1 to be $_, we can have shorter code. Since /\s+/ is the default pattern.

  ```
  $_ = "This is a project for CPS600.";
  @test2 = split;             # same as @test2 = split(/\s+/,$_)
  ```

- **The join() operator**

  The join() operator takes a list of values and combines them together with a glue string between each list element.

  ```
  $test1 = join(/ /, @test2);
  ```

  We can have the original $test1 by the join() operator. One thing we have to notice is the glue string is not a regular expression, it is just an ordinary string of zero or more characters.

---

8. Functions

We have seen some system functions such as print, split, join, sort, reverse, and so on. Let’s take a look at user defined functions.

Defining a User Function

A user function, usually called a subroutine or sub, is defined like:

```
sub subname {
    statement 1;
    statement 2;
    statement 3;
    statement 4;
        :
        :
        :
}
```

The subname is the name of the subroutine. It can be any name. The statements inside the block are the definitions of the subroutine. When a subroutine is called, the block of statements are executed and any return value is returned to the caller. Subroutine definitions can be put anywhere in the program. They will be skipped on execution. Subroutine definitions are global, there are no local subroutines. If you happen to have two subroutine definitions with the same name, the latter one will overwrite the former one without warning.

Invoking a User Function

How can we call a subroutine? We must precede the subroutine name with an ampersand(&) while you are trying to invoke a subroutine.

```
&say_hi;

sub say_hi {
   print "Say Hi to Neon!";
}
```

The result of this call will display "Say Hi to Neon!" on screen.

A subroutine can call another subroutine, and that subroutine can call another and so on until no memory left.

Return Values

Like in C, a subroutine is always part of some expression. The value of the subroutine invocation is called the *return value*. The rturn value of a subroutine is the value of the **last** expression **evaluated** within the body of the subroutine on each invocation.

```
$a = 5;
$b = 5;
$c = &sumab;                # $c is 10
$d = 5 + &sumab;            # $d is 15

sub sumab {
    $a + $b;
}
```

A subroutine can also return a list of values when evaluated in an array context.

```
$a = 3;
$b = 8;
@c = &listab;               # @c is (3, 8)

sub listab {
    ($a, $b);
}
```

The last expression evaluated means the last expression which is evaluated rather than the last expression defined in the subroutine. In the following example, the subroutine will return $a if $a > $b, otherwise, return $b.

```
sub choose_older {
    if ($a > $b) {
        print "Choose a\n";
        $a;
    } else {
        print "Choose b\n";
        $b;
    }
}
```

Arguments

The subroutine will be more helpful and useful if we can pass arguments. In Perl, if the subroutine invocation is followed by a list within parentheses, the list is automatically assigned to a special variable @_ for the duration of the subroutine. The subroutine can determine the number of arguments and the value of those arguments.

```
&say_hi_to("Neon");         # display "Say Hi to Neon!"
print &sum(3,8);            # display 11
$test = &sum(4,9);          # $test is 13

sub say_hi_to {
    print "Say Hi to $_[0]!\n";
}

sub sum {
    $_[0] + $_[1];
}
```

Excess parameters are ignored.

What if we want to add all of the elements in the list? Here is the example:

```
print &sum(1,2,3,4,5);      # display 15
print &sum(1,3,5,7,9);      # display 25
print &sum(1..10);          # display 55 since 1..10 is expanded

sub sum {
    $total = 0;
    foreach $_ (@_) {
        $total += $_;
    }
```

```
        $total;                 # last expression wvaluated
    }
```

Local Variables in Functions

We have know how to use @_ to invoke arguments in the subroutine. Now, you may want to create local versions of a list of variable names in the subroutine. You can do it by **local()** operator. Here is the sum subroutine with local() operator:

```
    sub sum {
        local($total);          # let $total be a local variable
        $total = 0;
        foreach $_ (@_) {
            $total += $_;
        }
        $total;                 # last expression wvaluated
    }
```

When the first body statement is executed, any current value of the global value $total is saved away and a new variable $total is created with an undef value. When the subroutine exits, Perl discards the local variable and restores the previous global value.

```
    sub larger_than {
        local($n, @list);
        ($n, @list) = @_;
        local(@result);
        foreach $_ (@list) {
            if ($_ > $n) {
                push(@result, $_);
            }
        }
        @result;
    }

    @test1 = &larger_than(25, 24, 43, 18, 27, 36);
    # @test1 gets (43,27,36)
    @test2 = &larger_than(12, 22, 33, 44, 11, 55, 3, 8);
    # @test2 gets (22,33,44,55)
```

We can also combine the first two lines of the above subroutine.

```
    local($n, @list) = @_;
```

This is, however, a common Perl like style. Here is a tip about the using of the local() operator. Try to put all of your local() operators at the beginning of the subroutine definition before you get into the main body of the subroutine.

---

9. Filehandles and File Tests

What is a Filehandle?

A *filehandle* is the name in a Perl program for an I/O connection between your Perl process and the outside world. Like block labels, filehandles are used without a special prefix character. It might be confused with some reserve words. Therefore, the inventor of Perl -- Larry Wall suggests people to use all **UPPERCASE** letters for the filehandle.

Opening and Closing a Filehandle

- **Opening a File**

  In Perl, we can use ***open()*** operator to open a filehandle. You can open a file for reading, writing or appending. Here are examples to do these actions:

  ```
  open(FILEHANDLE,"filename");
  # open a file for reading
  open(FILEHANDLE,">outputfile");
  # open a file for writing
  open(FILEHANDLE,">>appendfile");
  # open a file for appending
  ```

- **Closing a File**

  After you finish with a filehandle, you can use ***close()*** operator to close the filehandle. For example:

  ```
  close(FILEHANDLE);
  ```

Most of times, we want to make sure whether we have opened the file successfully or not. We can use the ***die()*** operator to inform us when the opening of a file fails. Usually, we use the following:

```
open(FILEHANDLE,"test") || die "Sorry! Cannot open the file "test".\n";
```

Using Filehandles

Once a filehandle is opened for reading, you can read lines from it just like you can read lines from < STDIN >. Same as < STDIN >, the newly opened filehandle must be in the angle brackets. Here is an example to copy a file to another file:

```
open(FILE1,$test1) || die "Cannot open $test1 for reading";
open(FILE2,">$test2") || die "Cannot create $b";
while (< FILE1 >) {         # read a line from file $test1 to $_
   print FILE2 $_;          # write the line into file $test2
}
close(FILE1);
close(FILE2);
```

File Tests

Sometimes, we may want to know if the file we are gonna process exists, or is readable or writable. We need the file tests to help us at this time. Here is a table containing file tests and their meaning:

File Test	Meaning
-r	File or directory is readable
-w	File or directory is writable
-x	File or directory is executable
-o	File or directory is owned by user
-R	File or directory is readable by real user
-W	File or directory is writable by real user
-X	File or directory is executable by real user
-O	File or directory is owned by real user
-e	File or directory exists
-z	File exists and has zero size
-s	File or directory exists and has nonzero size
-f	Entry is a plain file
-d	Entry is a directory
-l	Entry is a symlink
-S	Entry is a socket
-p	Entry is a named pipe (a "fifo")
-b	Entry is a block-special file (a mountable disk)
-c	Entry is a character-special file (an I/O device)
-u	File or directory is setuid
-g	File or directory is setgid
-k	File or directory has the sticky bit set
-t	isatty() on the filehandle is true
-T	File is "Text"
-B	File is "Binary"
-M	Modification age in days
-A	Access age in days
-C	Inode-modification age in days

You can check a list of filenames to see if they exist by the following method:

```
    foreach (@list_of_filenames) {
       print "$_ exists\n" if -e        # same as -e $_
    }
```

10. File and Directory Manioulation

Removing a File

Perl uses **unlink()** to delete files. Here are some examples:

```
    unlink("test");             # delete the file "test"
    unlink("test1","test2");    # delete 2 files "test1" and "test2"
    unlink(< *.ps >);           # delete all .ps files like "rm *.ps" in
                                  the shell
```

You can also provide the selection from the users.

```
    print "Input the filename you want to delete: ";
    chop($filename = < STDIN >);
    unlink($filename);
```

Renaming a File

We use ***mv*** to rename files in the shell. In Perl, we use ***rename($old, $new)***. For example:

```
$old = "test1";
$new = "test2";
rename($old, $new);         # "test1" is changed to "test2"
```

Creating Alternate Names for a File (Linking)

- **Hard Links**

  In the shell, we use ***ln old new*** to generate a hard link. In Perl, we use ***link("old", "new")*** to do it. However, there are some limitations to hard links. For a hard link, the old filename can not be a directory and the new alias must be on the same filesystem.

- **Symbolic Links(Symlinks or soft links)**

  In the shell, we use ***ln -s old new*** to get a symbolic link. In Perl, we use ***symlink("old", "new")***.

  When you invoke ***ls -l*** on the directory containing a symbolic link, you get an indication of both the name of the symbolic link and where the link points. Perl provides the same information by using ***readlink()***.

Making and Removing Directories

In the shell, we use ***mkdir*** command to make a directory. In Perl, similarly, it provides ***mkdir()*** operation. However, Perl adds one additional information in this operation. It can decide the permission at the same time. It takes two arguments: directory name and the permission. For example:

```
mkdir("test", 0755);        # It generates a directory called "test"
                              and its permission is "drwxr-xr-x"
```

You can use a ***rmdir(directory_name)*** to remove the directory just like ***rmdir directory_name*** in the shell.

Modifying Permissions

Just like ***chmod*** command in the shell, Perl has ***chmod()***. It takes two parts of arguments. The first part is the permission number (0644, 0755, ...) and the second part is a list of filenames. For example:

```
chmod(0644,"test1");        # change the permission of "test1" to be
                              "-rw-r--r--"
chmod(0644,"test1","test2");# change the permission of both files to
                              be "-rw-r--r--"
```

Modifying Ownership

Like ***chown*** in the shell, Perl has ***chown()*** operation. The chown() operator takes a user ID number(UID), a group ID number(GID) and a list of filenames. For example:

```
Assume "test"'s UID is 1234 and its GID is 56.

chown(1234, 56, "test1", "test2");
# make test1 and test2 belong to test and its default group.
```

---

11. Converting Other Languages to Perl

One of the great things of Perl is that there are some programs converting from different languages to Perl.

- ### Converting awk programs to Perl

  It can be done dy the *a2p* program provided with the Perl distribution. The usage is:

  ```
  $a2p < awkprog > perlprog
  ```

  Now, you can have the Perl program(script) ready to run.

- ### Converting sed programs to Perl

  This is similar to the previous one. Instead of using *a2p*, you can use *s2p* to convert sed programs to Perl programs.

- ### Converting shell programs to Perl

  Many people may ask: "What's about the shell programs?" However, there is **no** program converts shell programs to Perl programs. The best you can do is try to figure out the shell script and then start with Perl. You can, however, use a quick but dirty translation by putting the major portions of the original script inside the **system()** calls or backquotes. You may be able to replace some operations with native Perl. For example, replace **system("rm test")** with **unlink("test")**.

---

12. Glossaries and/or man pages

- Hypertext and searchable (courtesy of Robert Stockton at CMU)

- Hypertext Info file format
- Perl5 man page (at Nexor)
- Examples

    - Examples of the Programming Perl (Camel book)
    - Examples of the Learning Perl (Llama book)

Difference between Perl4 and Perl5

There exist Perl4 and Perl5. What's the difference between them? Basically, they are two different versions of the language. Perl5 has been modularized, object oriented, tweaked, trimmed, and optimized.

There are some new features shown in Perl5:

- **Enhanced usability**
- **Simplied grammar**

  The new yacc grammar is 1/2 of the old one. The reserved words are cut by 2/3.
- **Lexical scoping**
- **Arbitratrily nested data structures**
- **Modularity and reusability**
- **Object-oriented programming**

  A package can function as class. Filehandles are treated as objects.
- **Embeddible and extensible**

  Perl can be easily embedded in C/C++ applications. It can also either call or be called by your routines through a documentation interface. The XS preprocessor is provided to make it easy to glue your C/C++ routine into Perl. Dynamic loading of modules is supported.
- **POSIX compliant**

A new module -- POSIX module provides access to all available POSIX routines and definitions.

- **Package constructors and destructors**
- **Multiple simultaneous DBM implementations**
- **Subroutine definitions may be autoloaded**
- **Regular expression enhancements**

You can refer to **Metronet Perl5 Info** in **WWW sites for Perl** for more new features of Perl5.

WWW sites for Perl

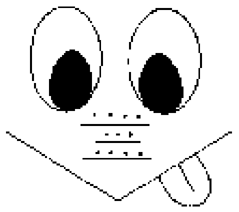
You can check the following WWW sites for more information:

1. University of Florida's Perl Archive
2. Metronet **Perl5** Info
3. Metronet Gopher Server
4. NEXOR Ltd Perl Page
5. Northwestern University
6. A Wais index of Comp.Lang.Perl
7. An Index of Perl/HTML archives
8. Neil Bowers's Perl page
9. Robert Seymour's Perl page
10. The Taming of the Camel -- An Overview of Perl 5.0 by Larry Wall
11. Larry Wall's witticisms
12. Yahoo's Perl page
13. The Perl FTP Archive (University of Florida)

References

1. **Larry Wall & Randal L. Schwartz, "The Camel Book -- Programming Perl", O'Reilly & Associates**
2. **Randal L. Schwartz, "The Llama Book -- Learning Perl", O'Reilly & Associates**
3. **Ellie Quigley, "Perl by Example", New Jersey, Prentice Hall, 1995**
4. **comp.lang.perl FAQ**

Please let me know what you think after you take a look at my tutorial Perl page. Just click on my email address below, you can send me mail. Thank you!



My Trademark -- Smiling Face!

Name	**Chang-An Hsiao (Andrew)**		
Email	chsiao@gatekeeper.cb.att.com		
Address	2565 Jonathan Parkway		
	Reynoldsburg	OH	43068-5255
Phone Number	**Home**	**Work**	**Fax**
	(614)868-8360	(614)860-2123	(614)868-3606