Introduction to Perl

Instructor: Dr. Nicholas C. Maliszewskyj
Textbook: Learning Perl on Win32 Systems (Schwartz, Olson & Christiansen)
Resources:

Programming Perl (Wall, Christiansen, & Schwartz)
Perl in a Nutshell (Siever, Spainhour, & Patwardian)
Perl Mongers http://www.perl.org/
Comprehensive Perl Archive Network http://www.cpan.org

Introduction

What is Perl?

Depending on whom you ask, Perl stands for "Practical Extraction and Report Language" or "Pathologically Eclectic Rubbish Lister." It is a powerful glue language useful for tying together the loose ends of computing life.

History

Perl is the natural outgrowth of a project started by Larry Wall in 1986. Originally intended as a configuration and control system for six VAXes and six SUNs located on opposite ends of the country, it grew into a more general tool for system administration on many platforms. Since its unveiling to programmers at large, it has become the work of a large body of developers. Larry Wall, however, remains its principle architect.

Although the first platform Perl inhabited was UNIX, it has since been ported to over 70 different operating systems including, but not limited to, Windows 9x/NT/2000, MacOS, VMS, Linux, UNIX (many variants), BeOS, LynxOS, and QNX.

Uses of Perl

1. Tool for general system administration
2. Processing textual or numerical data
3. Database interconnectivity
4. Common Gateway Interface (CGI/Web) programming
5. Driving other programs! (FTP, Mail, WWW, OLE)

Philosophy & Idioms

The Virtues of a Programmer

Perl is a language designed to cater to the three chief virtues of a programmer.

- Laziness - develop reusable and general solutions to problems
- Impatience - develop programs that anticipate your needs and solve problems for you.
- Hubris - write programs that you want other people to see (and be able to maintain)

There are many means to the same end

Perl provides you with more than enough rope to hang yourself. Depending on the problem, there may be several "official" solutions. Generally those that are approached using "Perl idioms" will be more efficient.

Resources

- The Perl Institute (http://www.perl.org)
- The Comprehensive Perl Archive Network (http://www.cpan.org)
- The Win32 port of Perl (http://www.activestate.com/ActivePerl/)

Perl Basics

Script names

While generally speaking you can name your script/program anything you want, there are a number of conventional extensions applied to portions of the Perl bestiary:

`.pm`	- Perl modules
`.pl`	- Perl libraries (and scripts on UNIX)
`.plx`	- Perl scripts

Language properties

- Perl is an interpreted language – program code is interpreted at run time. Perl is unique among interpreted languages, though. Code is compiled by the interpreter before it is actually executed.
- Many Perl idioms read like English
- Free format language – whitespace between tokens is optional
- Comments are single-line, beginning with **`#`**
- Statements end with a semicolon (**`;`**)
- Only subroutines and functions need to be explicitly declared
- Blocks of statements are enclosed in curly braces **`{}`**
- A script has no "**`main()`**"

Invocation

On platforms such as UNIX, the first line of a Perl program should begin with

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

and the file should have executable permissions. Then typing the name of the script will cause it to be executed.

Unfortunately, Windows does not have a real equivalent of the UNIX "shebang" line. On Windows 95/98, you will have to call the Perl interpreter with the script as an argument:

```
> perl myscript.plx
```

On Windows NT, you can associate the .plx extension with the Perl interpreter:

```
> assoc .plx=Perl
> ftype Perl=c:\myperl\bin\perl.exe %1% %*
> set PATHEXT=%PATHEXT%;.plx
```

After taking these steps, you can execute your script from the command line as follows:

```
> myscript
```

The *ActivePerl* distribution includes a **`pl2bat`** utility for converting Perl scripts into batch files.

You can also run the interpreter by itself from the command line. This is often useful to execute short snippets of code:

```
perl –e 'code'
```

Alternatively, you can run the interpreter in "debugging" mode to obtain a shell-like environment for testing code scraps:

```
perl –de 1
```

Data Types & Variables

Basic Types

The basic data types known to Perl are scalars, lists, and hashes.

Scalar	**`$foo`**	Simple variables that can be a number, a string, or a reference. A scalar is a "thingy."
List	**`@foo`**	An ordered array of scalars accessed using a numeric subscript. **`$foo[0]`**
Hash	**`%foo`**	An unordered set of key/value pairs accessed using the keys as subscripts. **`$foo{key}`**

Perl uses an internal type called a *typeglob* to hold an entire symbol table entry. The effect is that scalars, lists, hashes, and filehandles occupy separate namespaces (i.e., **`$foo[0]`** is not part of **`$foo`** or of **`%foo`**). The prefix of a typeglob is **`*`**, to indicate "all types." Typeglobs are used in Perl programs to pass data types by reference.

You will find references to literals and variables in the documentation. *Literals* are symbols that give an actual value, rather than represent possible values, as do *variables*. For example in **`$foo = 1`**, **`$foo`** is a scalar variable and **`1`** is an integer literal.

Variables have a value of `undef` before they are defined (assigned). The upshot is that accessing values of a previously undefined variable will not (necessarily) raise an exception.

Variable Contexts

Perl data types can be treated in different ways depending on the context in which they are accessed.

Scalar	Accessing data items as scalar values. In the case of lists and hashes, $foo[0] and $foo{key}, respectively. Scalars also have numeric, string, and don't-care contexts to cover situations in which conversions need to be done.
List	Treating lists and hashes as atomic objects
Boolean	Used in situations where an expression is evaluated as true or false. (Numeric: 0=false; String: null=false, Other: undef=false)
Void	Does not care (or want to care) about return value
Interpolative	Takes place inside quotes or things that act like quotes

Special Variables (defaults)

Some variables have a predefined and special meaning to Perl. A few of the most commonly used ones are listed below.

`$_`	The default input and pattern-searching space
`$0`	Program name
`$$`	Current process ID
`$!`	Current value of `errno`
`@ARGV`	Array containing command-line arguments for the script
`@INC`	The array containing the list of places to look for Perl scripts to be evaluated by the `do`, `require`, or `use` constructs
`%ENV`	The hash containing the current environment
`%SIG`	The hash used to set signal handlers for various signals

Scalars

Scalars are simple variables that are either numbers or strings of characters. Scalar variable names begin with a dollar sign followed by a letter, then possibly more letters, digits, or underscores. Variable names are case-sensitive.

Numbers

Numbers are represented internally as either signed integers or double precision floating point numbers. Floating point literals are the same used in C. Integer literals include decimal (255), octal (0377), and hexadecimal (0xff) values.

Strings

Strings are simply sequences of characters. String literals are delimited by quotes:

Single quote	**`‘string’`**	Enclose a sequence of characters
Double quote	**`“string”`**	Subject to backslash and variable interpolation
Back quote	**`` `command` ``**	Evaluates to the output of the enclosed command

The backslash escapes are the same as those used in C:

`\n`	Newline	**`\e`**	Escape
`\r`	Carriage return	**`\\`**	Backslash
`\t`	Tab	**`\”`**	Double quote
`\b`	Backspace	**`\’`**	Single quote

In Windows, to represent a path, use either “**`c:\\temp`**” (an escaped backslash) or “**`c:/temp`**” (UNIX-style forward slash).

Strings can be concatenated using the “**`.`**” operator: **`$foo = “hello” . ”world”;`**

Basic I/O

The easiest means to get operator input to your program is using the “diamond” operator:

```
$input = <>;
```

The input from the diamond operator includes a newline (\n). To get rid of this pesky character, use either **`chop()`** or **`chomp()`**. **`chop()`** removes the last character of the string, while **`chomp()`** removes *any* line-ending characters (defined in the special variable **`$/`**). If no argument is given, these functions operate on the **`$_`** variable.

To do the converse, simply use Perl’s print function:

```
print $output.”\n”;
```

Basic Operators

Arithmetic

Example	Name	Result
`$a + $b`	Addition	Sum of **`$a`** and **`$b`**
`$a * $b`	Multiplication	Product of **`$a`** and **`$b`**
`$a % $b`	Modulus	Remainder of **`$a`** divided by **`$b`**
`$a ** $b`	Exponentiation	**`$a`** to the power of **`$b`**

String

Example	Name	Result
`$a . “string”`	Concatenation	String built from pieces
`“$a string”`	Interpolation	String incorporating the value of **`$a`**
`$a x $b`	Repeat	String in which **`$a`** is repeated **`$b`** times

Assignment

The basic assignment operator is “**`=`**”: **`$a = $b`**.
Perl conforms to the C idiom that lvalue operator= expression
is evaluated as: lvalue = lvalue operator expression

So that	**`$a *= $b`**	is equivalent to	**`$a = $a * $b`**
	`$a += $b`		**`$a = $a + $b`**

This also works for the string concatenation operator: **`$a .= “\n”`**

Autoincrement and Autodecrement

The autoincrement and autodecrement operators are special cases of the assignment operators, which add or subtract 1 from the value of a variable:

`++$a, $a++`	Autoincrement	Add 1 to $a
`--$a, $a--`	Autodecrement	Subtract 1 from $a

Logical

Conditions for truth:

Any string is true except for “” and “0”
Any number is true except for 0
Any reference is true
Any undefined value is false

Example	Name	Result
`$a && $b`	And	True if both **`$a`** and **`$b`** are true
`$a || $b`	Or	**`$a`** if **`$a`** is true; **`$b`** otherwise
`!$a`	Not	True if **`$a`** is not true
`$a and $b`	And	True if both **`$a`** and **`$b`** are true
`$a or $b`	Or	**`$a`** if **`$a`** is true; **`$b`** otherwise
`not $a`	Not	True if **`$a`** is not true

Logical operators are often used to “short circuit” expressions, as in:

```
open(FILE,”< input.dat”) or die “Can’t open file”;
```

Comparison

Comparison	Numeric	String	Result
Equal	**`==`**	**`eq`**	True if $a equal to $b
Not equal	**`!=`**	**`ne`**	True if $a not equal to $b
Less than	**`<`**	**`lt`**	True if $a less than $b
Greater than	**`>`**	**`gt`**	True if $a greater than $b
Less than or equal	**`<=`**	**`le`**	True if $a not greater than $b
Comparison	**`<=>`**	**`cmp`**	0 if $a and $b equal 1 if $a greater -1 if $b greater

Operator Precedence

Perl operators have the following precedence, listed from the highest to the lowest, where operators at the same precedence level resolve according to associativity:

Associativity	Operators	Description
Left	Terms and list operators	
Left	`->`	Infix dereference operator
	`++`	Auto-increment
	`--`	Auto-decrement
Right	`\`	Reference to an object (unary)
Right	`! ~`	Unary negation, bitwise complement
Right	`+ -`	Unary plus, minus
Left	`=~`	Binds scalar to a match pattern
Left	`!~`	Same, but negates the result
Left	`* / % x`	Multiplication, Division, Modulo, Repeat
Left	`+ - .`	Addition, Subtraction, Concatenation
Left	`>> <<`	Bitwise shift right, left
	`< > <= >=`	Numerical relational operators
	`lt gt le ge`	String relational operators
	`== != <=>`	Numerical comparison operators
	`eq ne cmp`	String comparison operators
Left	`&`	Bitwise AND
Left	`| ^`	Bitwise OR, Exclusive OR
Left	`&&`	Logical AND
Left	`||`	Logical OR
	`..`	In scalar context, range operator In array context, enumeration
Right	`?:`	Conditional (if ? then : else) operator
Right	`= += -= etc`	Assignment operators
Left	`,`	Comma operator, also list element separator
	`=>`	Same, enforces left operand to be string
Right	`not`	Low precedence logical NOT
Right	`and`	Low precedence logical AND
Right	`or xor`	Low precedence logical OR

Parentheses can be used to group an expression into a term.

A list consists of expressions, variables, or lists, separated by commas. An array variable or an array slice many always be used instead of a list.

Control Structures

Statement Blocks

A statement block is simply a sequence of statements enclose in curly braces:

```
{
     first_statement;
     second_statement;
     last_statement
}
```

Conditional Structures (If/elsif/else)

The basic construction to execute blocks of statements is the **`if`** statement. The **`if`** statement permits execution of the associated statement block if the test expression evaluates as true. It is important to note that unlike many compiled languages, it is necessary to enclose the statement block in curly braces, even if only one statement is to be executed.

The general form of an if/then/else type of control statement is as follows:

```
if (expression_one) {
     true_one_statement;
} elsif (expression_two) {
     true_two_statement;
} else {
     all_false_statement;
}
```

For convenience, Perl also offers a construct to test if an expression is false:

```
unless (expression) {
     false_statement;
} else {
     true_statement;
}
```

Note that the order of the conditional can be inverted as well:

```
statement if (expression);
statement unless (expression);
```

The “ternary” operator is another nifty one to keep in your bag of tricks:

```
$var = (expression) ? true_value : false_value;
```

It is equivalent to:

```
if  (expression) {
     $var = true_value;
} else {
     $var = false_value;
}
```

Loops

Perl provides several different means of repetitively executing blocks of statements.

While

The basic *while* loop tests an expression before executing a statement block

```
while (expression) {
     statements;
}
```

Until

The *until* loop tests an expression at the end of a statement block; statements will be executed until the expression evaluates as true.

```
until (expression) {
     statements;
}
```

Do while

A statement block is executed at least once, and then repeatedly until the test expression is false.

```
do {
     statements;
} while (expression);
```

Do until

A statement block is executed at least once, and then repeatedly until the test expression is true.

```
do {
     statements;
} until (expression);
```

For

The *for* loop has three semicolon-separated expressions within its parentheses. These expressions function respectively for the initialization, the condition, and re-initialization expressions of the loop. The for loop

```
     for (initial_exp; test_exp; reinit_exp) {
          statements;
     }
```

This structure is typically used to iterate over a range of values. The loop runs until the **`test_exp`** is false.

```
     for ($i; $i<10;$i++) {
          print $i;
     }
```

Foreach

The *foreach* statement is much like the *for* statement except it loops over the elements of a list:

```
     foreach $i (@some_list) {
          statements;
     }
```

If the scalar loop variable is omitted, **`$_`** is used.

Labels

Any statement block can be given a label. Labels are identifiers that follow variable naming rules. They are placed immediately before a statement block and end with a colon:

```
     SOMELABEL: {
          statements;
     }
```

You can short-circuit loop execution with the directives **`next`** and **`last`**:

- **`next`** skips the remaining statements in the loop and proceeds to the next iteration (if any)
- **`last`** immediately exits the loop in question
- **`redo`** jumps to the beginning of the block (restarting current iteration)

Next and last can be used in conjunction with a label to specify a loop by name. If the label is omitted, the presumption is that next/last refers to the innermost enclosing loop.

Usually deprecated in most languages, the goto expression is nevertheless supported by Perl. It is usually used in connection with a label

```
     goto LABEL;
```

to jump to a particular point of execution.

Indexed Arrays (Lists)

A list is an ordered set of scalar data. List names follow the same basic rules as for scalars. A reference to a list has the form **`@foo`**.

List literals

List literals consist of comma-separated values enclosed in parentheses:

```
(1,2,3)
(“foo”,4.5)
```

A range can be represented using a list constructor function (such as “**`..`**”):

```
(1..9) = (1,2,3,4,5,6,7,8,9)
($a..$b) = ($a, $a+1, … , $b-1,$b)
```

In the case of string values, it can be convenient to use the “quote-word” syntax

```
@a = (“fred”,”barney”,”betty”,”wilma”);
@a = qw( fred barney betty wilma );
```

Accessing List Elements

List elements are subscripted by sequential integers, beginning with 0

`$foo[5]` is the sixth element of **`@foo`**

The special variable **`$#foo`** provides the index value of the last element of **`@foo`**.

A subset of elements from a list is called a *slice*.

`@foo[0,1]` is the same as **`($foo[0],$foo[1])`**

You can also access slices of list literals:

```
@foo = (qw( fred barney betty wilma ))[2,3]
```

List operators and functions

Many list-processing functions operate on the paradigm in which the list is a stack. The highest subscript end of the list is the “top,” and the lowest is the bottom.

`push`	Appends a value to the end of the list **`push(@mylist,$newvalue)`**
`pop`	Removes the last element from the list (and returns it) **`pop(@mylist)`**
`shift`	Removes the first element from the list (and returns it) **`shift(@mylist)`**
`unshift`	Prepends a value to the beginning of the list **`unshift(@mylist,$newvalue)`**
`splice`	Inserts elements into a list at an arbitrary position **`splice(@mylist,$offset,$replace,@newlist)`**

The **`reverse`** function reverses the order of the elements of a list

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

The **`sort`** function sorts the elements of its argument as strings in ASCII order. You can also customize the sorting algorithm if you want to do something special.

```
    @x = sort(@y);
```

The **`chomp`** function works on lists as well as scalars. When invoked on a list, it removes newlines (record separators) from each element of its argument.

Associative Arrays (Hashes)

A *hash* (or associative array) is an unordered set of key/value pairs whose elements are indexed by their keys. Hash variable names have the form **`%foo`**.

Hash Variables and Literals

A literal representation of a hash is a list with an even number of elements (key/value pairs, remember?).

```
     %foo = qw( fred wilma barney betty );
     %foo = @foolist;
```

To add individual elements to a hash, all you have to do is set them individually:

```
     $foo{fred} = “wilma”;
     $foo{barney} = “betty”;
```

You can also access slices of hashes in a manner similar to the list case:

```
     @foo{“fred”,”barney”} = qw( wilma betty );
```

Hash Functions

The **`keys`** function returns a list of all the current keys for the hash in question.

```
     @hashkeys = keys(%hash);
```

As with all other built-in functions, the parentheses are optional:

```
     @hashkeys = keys %hash;
```

This is often used to iterate over all elements of a hash:

```
     foreach $key (keys %hash) {
          print $hash{$key}.”\n”;
     }
```

In a scalar context, the **`keys`** function gives the number of elements in the hash.

Conversely, the **`values`** function returns a list of all current *values* of the argument hash:

```
     @hashvals = values(%hash);
```

The **`each`** function provides another means of iterating over the elements in a hash:

```
     while (($key, $value) = each (%hash)) {
          statements;
     }
```

You can remove elements from a hash using the **`delete`** function:

```
     delete $hash{‘key’};
```

Pattern Matching

Regular Expressions

Regular expressions are patterns to be matched against a string. The two basic operations performed using patterns are matching and substitution:

Matching	***/pattern/***
Substitution	**s*/pattern/newstring/***

The simplest kind of regular expression is a literal string. More complicated expressions include *metacharacters* to represent other characters or combinations of them.

The **`[…]`** construct is used to list a set of characters (a *character class*) of which *one* will match. Ranges of characters are denoted with a hyphen (**`-`**), and a negation is denoted with a circumflex (**`^`**). Examples of character classes are shown below:

`[a-zA-Z]`	Any single letter
`[0-9]`	Any digit
`[^0-9]`	Any character **not** a digit

Some common character classes have their own predefined symbols:

Code	Matches
`.`	Any character
`\d`	A digit, such as **`[0-9]`**
`\D`	A nondigit, same as **`[^0-9]`**
`\w`	A word character (alphanumeric) **`[a-zA-Z_0-9]`**
`\W`	A nonword character **`[^a-zA-Z_0-9]`**
`\s`	A whitespace character **`[ \t\n\r\f]`**
`\S`	A non-whitespace character **`[^ \t\n\r\f]`**

Regular expressions also allow for the use of both variable interpolation and *backslashed representations* of certain characters:

Code	Matches
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\f`	Formfeed
`\/`	Literal forward slash

Anchors don't match any characters; they match places within a string.

Assertion	Meaning
`^`	Matches at the beginning of string
`$`	Matches at the end of string
`\b`	Matches on word boundary
`\B`	Matches except at word boundary
`\A`	Matches at the beginning of string
`\Z`	Matches at the end of string or before a newline
`\z`	Matches only at the end of string

Quantifiers are used to specify how many instances of the previous element can match.

Maximal	Minimal	Allowed Range
`{n,m}`	**`{n,m}?`**	Must occur at least **`n`** times, but no more than **`m`** times
`{n,}`	**`{n,}?`**	Must occur at least **`n`** times
`{n}`	**`{n}?`**	Must match exactly **`n`** times
`*`	**`*?`**	0 or more times (same as **`{0,}`**)
`+`	**`+?`**	1 or more times (same as **`{1,}`**)
`?`	**`??`**	0 or 1 time (same as **`{0,1}`**)

It is important to note that quantifiers are greedy by nature. If two quantified patterns are represented in the same regular expression, the leftmost is greediest. To force your quantifiers to be non-greedy, append a question mark.

If you are looking for two possible patterns in a string, you can use the alternation operator (**`|`**). For example,

```
/you|me|him|her/;
```

will match against any one of these four words. You may also use parentheses to provide boundaries for alternation:

```
/And(y|rew)/;
```

will match either “Andy” or “Andrew”.

Parentheses are used to group characters and expressions. They also have the effect of “remembering” parts of a matched pattern for further processing. To recall the “memorized” portion of the string, include a backslash followed by an integer representing the location of the parentheses in the expression:

```
/fred(.)barney\1/;
```

Outside of the expression, these “memorized” portions are accessible as the special variables **`$1`**, **`$2`**, **`$3`**, etc. Other special variables are as follows:

`$&`	Part of string matching regexp
`` $` ``	Part of string *before* the match
`$’`	Part of string *after* the match

Regular expression grouping precedence

Parentheses	**`() (?: )`**
Quantifiers	**`? + * {m,n} ?? +? *?`**
Sequence and anchoring	**`abc ^ $ \A \Z (?= ) (?! )`**
Alternation	**`|`**

To select a target for matching/substitution other than the default variable (**`$_`**), use the **`=~`** operator:

```
$var =~ /pattern/;
```

Operators

`m/pattern/gimosx`

The “match” operator searches a string for a pattern match. The preceding “m” is usually omitted. The trailing modifiers are as follows

Modifier	**Meaning**
`g`	Match globally; find all occurrences
`i`	Do case-insensitive matching
`m`	Treat string as multiple lines
`o`	Only compile pattern once
`s`	Treat string as a single line
`x`	Use extended regular expressions

`s/pattern/replacement/egimosx`

Searches a string for a pattern, and replaces any match with replacement. The trailing modifiers are all the same as for the match operator, with the exception of “**`e`**”, which evaluates the right-hand side as an expression. The substitution operator works on the default variable (**`$`**_), unless the **`=~`** operator changes the target to another variable.

`tr/pattern1/pattern2/cds`

This operator scans a string and, character by character, replaces any characters matching **`pattern1`** with those from **`pattern2`**. Trailing modifiers are:

Modifier	Meaning
`c`	Complement pattern1
`d`	Delete found but unreplaced characters
`s`	Squash duplicated replaced characters

This can be used to force letters to all uppercase:

```
tr/a-z/A-Z/;
```

`@fields = split(pattern,$input);`

Split looks for occurrences of a regular expression and breaks the input string at those points. Without any arguments, split breaks on the whitespace in $_:

`@words = split;` is equivalent to
`@words = split(/\s+/,$_);`

`$output = join($delimiter,@inlist);`

Join, the complement of split, takes a list of values and glues them together with the provided delimiting string.

Subroutines and Functions

Subroutines are defined in Perl as:

```
    sub subname {
         statement_1;
         statement_2;
    }
```

Subroutine definitions are global; there are no local subroutines.

Invoking subroutines

The ampersand (&) is the identifier used to call subroutines. They may also be called by appended parentheses to the subroutine name:

```
    name();
    &name;
```

You may use the explicit **`return`** statement to return a value and leave the subroutine at any point.

```
    sub myfunc {
         statement_1;
         if (condition) return $val;
         statement_2;
         return $val;
    }
```

Passing arguments

Arguments to a subroutine are passed as a single, flat list of scalars, and return values are passed the same way. Any arguments passed to a subroutine come in as `@_`.

To pass lists of hashes, it is necessary to pass *references* to them:

```
    @returnlist = ref_conversion(\@inlist, \%inhash);
```

The subroutine will have to *dereference* the arguments in order to access the data values they represent.

```
    sub myfunc {
         my($inlistref,$inhashref) = @_;
         my(@inlist) = @$inlistref;
         my(%inhash) = %$inhashref;
         statements;
         return @result;
    }
```

Prototypes allow you to design your subroutines to take arguments with constraints on the number of parameters and types of data.

Variable Scope

Any variables used in a subroutine that aren't declared private are global variables.

The **`my`** function declares variables that are lexically scoped within the subroutine. This means that they are private variables that only exist within the block or routine in which they are called. The **`local`** function declares variables that are dynamic. This means that they have global scope, but have temporary values within the subroutine. Most of the time, use **`my`** to localize variables within a subroutine.

Files and I/O

Filehandles

A filehandle is the name for the connection between your Perl program and the operating system. Filehandles follow the same naming conventions as labels, and occupy their own namespace.

Every Perl program has three filehandles that are automatically opened for it: STDIN, STDOUT, and STDERR:

STDIN	Standard input (keyboard or file)
STDOUT	Standard output (print and write send output here)
STDERR	Standard error (channel for diagnostic output)

Filehandles are created using the open() function:

```
open(FILE,”filename”);
```

You can open files for reading, writing, or appending:

`open(FILE,”>  newout.dat”)`	Writing, creating a new file
`open(FILE,”>> oldout.dat”)`	Appending to existing file
`open(FILE,”<  input.dat”)`	Reading from existing file

As an aside, under Windows, there are a number of ways to refer to the full path to a file:

`”c:\\temp\\file”`	Escape the backslash in double quotes
`‘c:\temp\file’`	Use proper path in single quotes
`“c:/temp/file”`	UNIX-style forward slashes

It is important to realize that calls to the **`open()`** function are not always successful. Perl will not (necessarily) complain about using a filehandle created from a failed **`open()`**. This is why we test the condition of the open statement:

```
open(F,”< badfile.dat”) or die “open: $!”
```

You may wish to test for the existence of a file or for certain properties before opening it. Fortunately, there are a number of file test operators available:

File test	Meaning
`-e file`	File or directory exists
`-T file`	File is a text file
`-w file`	File is writable
`-r file`	File is readable
`-s file`	File exists and has nonzero length

These operators are usually used in a conditional expression:

```
if (-e myfile.dat) {
     open(FILE,”< myfile.dat”) or die “open: $!\n”;
}
```

Even more information can be obtained about a given file using the **`stat()`** function.

Using filehandles

After a file has been opened for reading you can read from it using the diamond operator just as you have already done for STDIN:

```
$_ = <FILE>;          or
while (<FILE>) {
     statements;
}
```

To print to your open output file, use the filehandle as the first argument to the print statement (N.B. no commas between the filehandle and the string to print).

```
print FILE “Look Ma! No hands!\n”;
```

To change the default output filehandle from STDOUT to another one, use select:

```
select FILE;
```

From this point on, all calls to print or write without a filehandle argument will result in output being directed to the selected filehandle. Any special variables related to output will also then apply to this new default. To change the default back to STDOUT, select it:

```
select STDOUT;
```

When you are finished using a filehandle, close it using close():

```
close(FILE);
```

Formats

Perl has a fairly sophisticated mechanism for generating formatted output. This involves using pictorial representations of the output, called *templates*, and the function **`write`**. Using a format consists of three operations:

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

Format templates have the following general form:

```
format FORMATNAME =
fieldline
$value_one, $value_two, …
.
```

Everything between the "=" and the "." is considered part of the format and everything (in the *fieldlines*) will be printed exactly as it appears (whitespace counts). Fieldlines permit variable interpolation via *fieldholders*:

```
Hi, my name is @<<<<<, and I'm @< years old.
$name, $age
```

Fieldline
Valueline

Fieldholders generally begin with a `@` and consist of characters indicated alignment/type.

`@<<<`	Four character, left-justified field
`@>>>`	Four character, right-justified field
`@|||`	Four character, centered field
`@###.##`	Six character numeric field, with two decimal places
`@*`	Multi-line field (on line by itself – for blocks of text)
`^<<<<`	Five character, "filled" field ("chews up" associated variables)

The name of the format corresponds to the name of a filehandle. If write is invoked on a filehandle, then the corresponding format is used. Naturally then, if you're printing to standard output, then your format name should be STDOUT. If you want to use a format with a name other than that of your desired filehandle, set the **`$~`** variable to the format name.

There are special formats which are printed at the top of the page. If the active format name is FNAME, then the "top" format name is FNAME_TOP. The special variable **`$%`** keeps a count of how many times the "top" format has been called and can be used to number pages.

Manipulating files & directories

The action of opening a file for writing creates it. Perl also provides functions to manipulate files without having to ask the operating system to do it.

`unlink(filename)`

Delete an existing file. Unlink can take a list of files, or wildcard as an argument as well: unlink(<*.bak>)

`rename(oldname, newname)`

This function renames a file. It is possible to move files into other directories by specifying a path as part of the new name.

Directories also have some special function associated with them

`mkdir(dirname, mode)`

Create a new directory. The “mode” specifies the permissions (set this to 0777 to be safe).

`rmdir(dirname)`

Removes (empty) directories

`chdir(dirname)`

Change current working directory to dirname

File and directory attributes can be modified as well:

`chmod(permission, list of files)`

Change the permissions of files or directories:

666 = read and write
444 = read only
777 = read, write, and executable

`utime(atime, mtime, list of files)`

Modify timestamps on files or directories. “atime” is the time of the most recent access, and “mtime” is the time the file/directory was last modified.

Modules

Namespaces and Packages

Namespaces store identifiers for a package, including variables, subroutines, filehandles, and formats, so that they are distinct from those of another package. The default namespace for the body of any Perl program is `main`. You can refer to the variables from another package by "qualifying" them with the package name. To do this, place the name of the package followed by two colons before the identifier's name:

```
$Package::varname
```

If the package name is null, the `main` package is assumed.

Modules

Modules are Perl's answer to software packages. They extend the functionality of core Perl with additional compiled code and scripts. To make use of a package (if it's installed on your system), call the use function:

```
use CGI;
```

This will pull in the module's subroutines and variables at compile time. **`use`** can also take a list of strings naming entities to be imported from the module:

```
use Module qw(const1 const2 func1 func2);
```

Perl looks for modules by searching the directories listed in **`@INC`**.

Modules can be obtained from the Comprehensive Perl Archive Network (CPAN) at
    http://www.cpan.org/modules/
or from the ActiveState site:
    http://www.ActiveState.com/packages/zips/

To install modules under UNIX, unarchive the file containing the package, change into its directory and type:

```
perl Makefile.PL
make
make install
```

On Windows, the ActivePerl distribution makes use of the "Perl Package Manager" to install/remove/update packages. To install a package, run ppm on the .ppd file associated with the module:

```
ppm install module.ppd
```

Object-Oriented Perl

In Perl, modules and object-oriented programming go hand in hand. Not all modules are written in an object-oriented fashion, but most are. A couple of definitions are warranted here:

- An *object* is simply a referenced thingy that happens to know which class it belongs to.
- A *class* is simply a package that happens to provide methods to deal with objects.
- A *method* is simply a subroutine that expects an object reference (or a package name, for class methods) as its first argument.

To create an object (or instance of a class), use the class constructor. Usually the class constructor will be a function named “new,” but may be called “Create” for some Win32 modules. For example,

```
$tri = new Triangle::Right (side1=>3, side2=>4);
```

The constructor takes a list of arguments describing the properties of the object to be created (see the documentation of the module in question to determine what these should be) and returns a reference to the created object.

An example of a class constructor (internal to the module) is shown below:

```
package critter; # declare the name of the package

sub new {
      my $class = shift;   # Get class name
      my $self = {};       # Initialize the object to nothing
      bless $self, $class; # Declare object to be part of class
      $self->_initialize();# Do other initializations
      return $self;
}
```

Methods (subroutines expecting an object reference as their first argument) may be invoked in two ways:

```
Packagename->constructor(args)->method_name(args)
```

Or:

```
$object = Packagename->constructor(args);
$object->method_name(args);
```

Methods are simply declared subroutines in the package source file.

Common Gateway Interfaces

Perl is the most commonly used language for CGI programming on the World Wide Web. The Common Gateway Interface (CGI) is an essential tool for creating and managing comprehensive websites. With CGI, you can write scripts that create interactive, user-driven applications.

Simple CGI Programs

CGI programs are invoked by accessing a URL (uniform resource locator) describing their coordinates:

```
http://www.mycompany.com/cgi-bin/program.plx
```

even though the actual location of the script on the hard drive of the server might be something like:

```
c:\webserver\bin\program.plx
```

The simplest CGI programs merely write HTML data to STDOUT (which is then displayed by your browser):

```
print << ENDOFTEXT;
Content-type: text/html

<HTML>
<HEAD><TITLE>Hello, World!</TITLE></HEAD>
<BODY>
<H1>Hello, World!</H1>
</BODY>
</HTML>
ENDOFTEXT
```

CGI.pm

CGI.pm is a Perl module written to facilitate CGI programming. It contains within itself the wherewithal to generate HTML, parse arguments and environment variables, and maintain state over multiple transactions. Another feature which is not to be underestimated is the ability to reliably run CGI programs from the command line for the purposes of debugging. A simple example of the CGI.pm module in action is shown below:

```
use CGI;

$query = CGI::new();    # create CGI object
$bday = $query->param(“birthday”); # get parameters
print $query->header(); # generate Content-type line
print $query->p(“Your birthday is $bday”);
```

Passing Parameters

CGI programs really shine, however, when they take arguments and format their output depending on those arguments. To pass arguments to a CGI script, ampersand-delimited key-value pairs to the URL:

```
http://www.mycompany.com/cgi-bin/icecream.plx?flavor=mint&container=cone
```

Everything after the question mark is an argument to the script.

Environment variables provide the script with data about the server, client, and the arguments to the script. The latter are available as the "QUERY_STRING" environment variable. The following example prints all of the environment variables:

```
# Print all of the necessary header info
print <<ENDOFTEXT;
Content-type: text/html

<HTML>
<HEAD><TITLE>Environment Variables</TITLE></HEAD>
<BODY>
<CENTER><H1>Environment Variables</H1></CENTER>
<TABLE>
<TR><TH>Variable</TH><TH>Value</TH>
ENDOFTEXT

# Now loop over and format environment variables
foreach $key (sort keys %ENV) {
    print "<TR><TD>$key</TD><TD>$ENV{$key}</TD></TR>\n";
}
print "</TABLE></BODY></HTML>\n";
```

CGI.pm is particularly good at extracting parameters in a platform-independent way:

```
use CGI;

$query = CGI::new();
print $query->header();
print $query->start_html(-title=>'Form Parameters');
print $query->h1('Form Parameters');
foreach $name ( $query->param() ) {
    $value = $query->param($name);
    print "$name => $value\n";
    print $query->br();        # Insert a line break

}
print $query->end_html();
```

Database Access

There are two primary means of accessing databases under Perl. The first (and oldest) makes use of the DBM (Database Management) libraries available for most flavors of UNIX. The second (and more powerful) allows for a platform-independent interaction with more sophisticated database management systems (DBMS's) such as Oracle, Sybase, Informix, and MySQL.

DBM

A DBM is a simple database management facility for most UNIX systems. It allows programs to store a collection of key-value pairs in binary form, thus providing rudimentary database support for Perl. To use DBM databases in Perl, you can associate a hash with a DBM database through a process similar to opening a file:

```
use DB_File;
tie(%ARRAYNAME, "DB_File", "dbmfilename");
```

Once the database is opened, anything you do to the hash is immediately written to the database. To break the association of the hash with the file, use the `untie()` function.

DBI/DBD

DBI is a module that provides a consistent interface for interaction with database solutions. The DBI approach relies on database-specific drivers (DBD's) to translate the DBI calls as needed for each database. Further, actual manipulation of the contents of the database is performed by composing statements in Structured Query Language (SQL) and submitting them to the database server.

DBI methods make use of two different types of handles

1. Database handles (like filehandles)
2. Statement handles (provide means of executing statements and manipulating their results)

Database handles are created by the `connect()` method:

```
$db_handle = DBI->connect('DBI:mysql:dbname:hostname',
                          $username, $password);
```

and destroyed by the `disconnect()` method:

```
$result = $db_handle->disconnect();
```

The first argument to the `connect()` method is a string describing the data source, typically written in the form:

```
DBI:driver_name:database_name:host_name
```

Statement handles are created by the `prepare()` method

```
$st_handle = $db_handle->prepare($sql)
```

where `$sql` is a valid SQL statement, and "destroyed" using the `finish()` method.

The SQL statement is then executed using the `execute()` method

```
$result = $st_handle->execute();
```

and the results obtained using any of the `fetch()` methods:

```
@ary = $st_handle->fetchrow_array(); # fetch a single row of the
                                     # query results

$hashref = $st_handle->fetchrow_hashref();
%hash = %$hashref;
```

Note that you do not directly access the results the SQL statement, but obtain them one row at a time via the `fetch()` methods.

The following script connects to a MySQL database and prints the contents of one of its tables:

```
use DBI:
use strict:

my($dsn) = ‘DBI:mysql:test:localhost’; # Data source name
my($username) = ‘user’;     # User name
my($password) = ‘secret’;   # Password
my($dbh,$sth);              # Database and statement handles
my(@ary);                   # array for rows returned by query

# connect to database
$dbh = DBI->connect($dsn, $username, $password);

# issue query
$sth = $dbh->prepare(‘SELECT * FROM tablename’);
$sth->execute();

# read results of query, then clean up
while(@ary = $sth->fetchrow_array()) {
    print join(“\t”, @ary), “\n”;
}
$sth->finish();

$dbh->disconnect();
```