4

The Perl Language

This chapter is a quick and merciless guide to the Perl language itself. If you're trying to learn Perl from scratch and would prefer to be taught rather than to have things thrown at you, then you might be better off with *Learning Perl, 3rd Edition* by Randal L. Schwartz and Tom Phoenix. However, if you already know some other programming languages and just want to learn the particulars of Perl, this chapter is for you. Sit tight, and forgive us for being terse—we have a lot of ground to cover.

If you want a more complete discussion of the Perl language and its idiosyncrasies (and we mean *complete*), see *Programming Perl, 3rd Edition* by Larry Wall, Tom Christiansen, and Jon Orwant.

Program Structure

Perl is a particularly forgiving language, as far as program layout goes. There are no rules about indentation, newlines, etc. Most lines end with semicolons, but not everything has to. Most things don't have to be declared, except for a couple of things that do. Here are the bare essentials:

Whitespace
: Whitespace is required only between items that would otherwise be confused as a single term. All types of whitespace—spaces, tabs, newlines, etc.—are equivalent in this context. A comment counts as whitespace. Different types of whitespace are distinguishable within quoted strings, formats, and certain line-oriented forms of quoting. For example, in a quoted string, a newline, a space, and a tab are interpreted as unique characters.

Semicolons
: Every simple statement must end with a semicolon. Compound statements contain brace-delimited blocks of other statements and do not require

terminating semicolons after the ending brace. A final simple statement in a block also does not require a semicolon.

Declarations
: Only subroutines and report formats need to be explicitly declared. All other user-created objects are automatically created with a null or 0 value unless they are defined by some explicit operation such as assignment. The *–w* command-line switch will warn you about using undefined values.

  You may force yourself to declare your variables by including the `use strict` pragma in your programs (see Chapter 8 for more information on pragmas and `strict` in particular). This causes an error if you do not explicitly declare your variables.

Comments and documentation
: Comments within a program are indicated by a pound sign (`#`). Everything following a pound sign to the end of the line is interpreted as a comment.

  Lines starting with `=` are interpreted as the start of a section of embedded documentation (pod), and all subsequent lines until the next `=cut` are ignored by the compiler. See the section "Formats" later in this chapter for more information on pod format.

Data Types and Variables

Perl has three basic data types: *scalars*, *arrays*, and *hashes*.

Scalars are essentially simple variables. They are preceded by a dollar sign (`$`). A scalar is either a number, a string, or a reference. (A reference is a scalar that points to another piece of data. References are discussed later in this chapter.) If you provide a string in which a number is expected or vice versa, Perl automatically converts the operand using fairly intuitive rules.

Arrays are ordered lists of scalars accessed with a numeric subscript (subscripts start at 0). They are preceded by an "at" sign (`@`).

Hashes are unordered sets of key/value pairs accessed with the keys as subscripts. They are preceded by a percent sign (`%`).

Numbers

Perl stores numbers internally as either signed integers or double-precision, floating-point values. Numeric literals are specified by any of the following floating-point or integer formats:

`12345`
: Integer

`-54321`
: Negative integer

`12345.67`
: Floating point

`6.02E23`
: Scientific notation

`0xffff`
: Hexadecimal

`0377`
: Octal

`4_294_967_296`
: Underline for legibility

Since Perl uses the comma as a list separator, you cannot use a comma for improving the legibility of a large number. To improve legibility, Perl allows you to use an underscore character instead. The underscore works only within literal numbers specified in your program, not in strings functioning as numbers or in data read from somewhere else. Similarly, the leading `0x` for hex and `0` for octal work only for literals. The automatic conversion of a string to a number does not recognize these prefixes—you must do an explicit conversion.

Be aware that in Perl 5.8, there are many changes in how Perl deals with integers and floating-point numbers. Regardless of how your system handles numbers and conversion between characters and numbers, Perl 5.8 works around system deficiencies to force more accurate number handling. Furthermore, whereas prior to 5.8 Perl used floating-point numbers exclusively in math operations, Perl 5.8 now uses and stores integers in numeric conversions and in arithmetic operations.

String Interpolation

Strings are sequences of characters. String literals are usually delimited by either single (`'`) or double (`"`) quotes. Double-quoted string literals are subject to backslash and variable interpolation, and single-quoted strings are not (except for `\'` and `\\`, used to put single quotes and backslashes into single-quoted strings). You can embed newlines directly in your strings.

Table 4-1 lists all the backslashed or escape characters that can be used in double-quoted strings.

Table 4-1. Double-quoted string representations

Code	Meaning
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\f`	Form feed
`\b`	Backspace
`\a`	Alert (bell)
`\e`	ESC character
`\033`	ESC in octal
`\x7f`	DEL in hexadecimal
`\cC`	Ctrl-C

Table 4-1. Double-quoted string representations (continued)

Code	Meaning
\\	Backslash
\"	Double quote
\u	Force next character to uppercase
\l	Force next character to lowercase
\U	Force all following characters to uppercase
\L	Force all following characters to lowercase
\Q	Backslash all following non-alphanumeric characters
\E	End \U, \L, or \Q

Table 4-2 lists alternative quoting schemes that can be used in Perl. These are useful in diminishing the number of commas and quotes you may have to type, and they allow you not to worry about escaping characters such as backslashes when there are many instances in your data. The generic forms allow you to use any non-alphanumeric, non-whitespace characters as delimiters in place of the slash (/). If the delimiters are single quotes, no variable interpolation is done on the pattern. Parentheses, brackets, braces, and angle brackets can be used as delimiters in their standard opening and closing pairs.

Table 4-2. Quoting syntax in Perl

Customary	Generic	Meaning	Interpolation
''	q//	Literal	No
""	qq//	Literal	Yes
``	qx//	Command	Yes
( )	qw//	Word list	No
( )	qr//	Pattern	Yes
//	m//	Pattern match	Yes
s///	s///	Substitution	Yes
y///	tr///	Translation	No

Here Documents

A line-oriented form of quoting is based on the Unix shell "here-document" syntax. Following a <<, you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. This is of particular importance if you're trying to print something like HTML that would be cleaner to print as a chunk instead of as individual lines. For example:

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

my $Price = 'right';

print <<"EOF";
```

```
The price is $Price.
EOF
```

The terminating string does not have to be quoted. For example, the previous example could have been written as:

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

my $Price = 'right';

print <<EOF;
The price is $Price.
EOF
```


You can assign here documents to a string:

```
my $assign_this_heredoc =<< "EOS";
This string is assigned to $whatever.
EOS
```

You can use a here document to execute commands:

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

print <<`CMD`;
ls -l
CMD
```

You can stack here documents:

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

print <<"joe", <<"momma"; # You can stack them
I said foo.
joe
I said bar.
momma
```

One caveat about here documents: you may have noticed in each of these examples that the quoted text is always left-justified. That's because any whitespace used for indentation will be included in the string. For example:

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

print <<"    INDENTED";
    Same old, same old.
    INDENTED
```

Although you can use a trick of including whitespace in the terminating tag to keep it indented (as we did here), the string itself will have the whitespace embedded—in this case, it will be `Same old, same old.`.

Lists

A list is an ordered group of scalar values. A literal list can be composed as a comma-separated list of values contained in parentheses, for example:

```
(1,2,3)                 # Array of three values 1, 2, and 3
("one","two","three")   # Array of three values "one", "two", and "three"
```

The generic form of list creation uses the quoting operator qw// to contain a list of values separated by whitespace:

```
qw/snap crackle pop/
```

With the quoting operators, you're not limited to // when you use one of the operators. You can use just about any character you want. The following is exactly the same as the example above:

```
qw!snap crackle pop!
```

It's important that you remember not to use any delimiters except whitespace with qw//. If you do, these delimiters will be handled as list members:

```
@foods = qw/fish, beef, lettuce, cat, apple/; # EL WRONG-O!
foreach (@foods) {
    print $_; # Prints fish and then a literal comma, etc.
}
```

Variables

A variable always begins with the character that identifies its type: $, @, or %. Most of the variable names you create can begin with a letter or underscore, followed by any combination of letters, digits, or underscores, up to 255 characters in length. Upper- and lowercase letters are distinct. Variable names that begin with a digit can contain only digits, and variable names that begin with a character other than an alphanumeric or underscore can contain only that character. The latter forms are usually predefined variables in Perl, so it is best to name your variables beginning with a letter or underscore.

Variables have the undef value before they are first assigned or when they become "empty." For scalar variables, undef evaluates to 0 when used as a number, and a zero-length, empty string ("") when used as a string.

Simple variable assignment uses the assignment operator (=) with the appropriate data. For example:

```
$age = 26;               # Assigns 26 to $age
@date = (8, 24, 70);     # Assigns the three-element list to @date
%fruit = ('apples', 3, 'oranges', 6);
 # Assigns the list elements to %fruit in key/value pairs
```

Scalar variables are always named with an initial $, even when referring to a scalar value that is part of an array or hash.

Every variable type has its own namespace. You can, without fear of conflict, use the same name for a scalar variable, an array, or a hash (or, for that matter, a filehandle, a subroutine name, or a label). This means that $foo and @foo are two different variables. It also means that $foo[1] is an element of @foo, not a part of $foo.

Arrays

An array is a variable that stores an ordered list of scalar values. Arrays are preceded by an "at" sign (@).

```
@numbers = (1,2,3);        # Set the array @numbers to (1,2,3)
```

To refer to a single element of an array, use the dollar sign ($) with the variable name (it's a scalar), followed by the index of the element in square brackets (the *subscript operator*). Array elements are numbered starting at 0. Negative indexes count backwards from the last element in the list (i.e., –1 refers to the last element of the list). For example, in this list:

```
@date = (8, 24, 70);
```

`$date[2]` is the value of the third element, 70.

Hashes

A hash is a set of key/value pairs. Hashes are preceded by a percent sign (%). To refer to a single element of a hash, you use the hash variable name followed by the "key" associated with the value in braces. For example, the hash:

```
%fruit = ('apples', 3, 'oranges', 6);
```

has two values (in key/value pairs). If you want to get the value associated with the key `apples`, you use `$fruit{'apples'}`.

It is often more readable to use the => operator in defining key/value pairs. The => operator is similar to a comma, but it's more visually distinctive and quotes any bare identifiers to the left of it:

```
%fruit = (
    apples  => 3,
    oranges => 6
);
```

Scalar and List Contexts

Every operation that you invoke in a Perl script is evaluated in a specific context, and how that operation behaves may depend on the context it is being called in. There are two major contexts: *scalar* and *list*. All operators know which context they are in, and some return lists in contexts wanting a list and scalars in contexts wanting a scalar. For example, the `localtime` function returns a nine-element list in list context:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime( );
```

But in a scalar context, `localtime` returns the number of seconds since January 1, 1970:

```
$now = localtime( );
```

Statements that look confusing are easy to evaluate by identifying the proper context. For example, assigning what is commonly a list literal to a scalar variable:

```
$a = (2, 4, 6, 8);
```

gives `$a` the value 8. The context forces the right side to evaluate to a scalar, and the action of the comma operator in the expression (in the scalar context) returns the value farthest to the right.

Another type of statement that might be confusing is the evaluation of an array or hash variable as a scalar. For example:

```
$b = @c;
```

When an array variable is evaluated as a scalar, the number of elements in the array is returned. This type of evaluation is useful for finding the number of elements in an array. The special `$#array` form of an array value returns the index of the last member of the list (one less than the number of elements).

If necessary, you can force a scalar context in the middle of a list by using the `scalar` function.

Declarations and Scope

In Perl, only subroutines and formats require explicit declaration. Variables (and similar constructs) are automatically created when they are first assigned.

Variable declaration comes into play when you need to limit the scope of a variable's use. You can do this in two ways:

Dynamic scoping
: Creates temporary objects within a scope. Dynamically scoped constructs are visible globally, but take action only within their defined scopes. Dynamic scoping applies to variables declared with `local`.

Lexical scoping
: Creates private constructs that are visible only within their scopes. The most frequently seen form of lexically scoped declaration is the declaration of `my` variables.

Therefore, we can say that a `local` variable is *dynamically scoped*, whereas a `my` variable is *lexically scoped*. Dynamically scoped variables are visible to functions called from within the block in which they are declared. Lexically scoped variables, on the other hand, are totally hidden from the outside world, including any called subroutines, unless they are declared within the same scope. See the section "Subroutines" later in this chapter for further discussion.

Statements

A simple statement is an expression evaluated for its side effects. Every simple statement must end in a semicolon, unless it is the final statement in a block.

A sequence of statements that defines a scope is called a *block*. Generally, a block is delimited by braces, or `{ }`. Compound statements are built out of expressions and blocks. A conditional expression is evaluated to determine whether a statement block will be executed. Compound statements are defined in terms of blocks, not statements, which means that braces are required.

Any block can be given a label. *Labels* are identifiers that follow the variable-naming rules (i.e., they begin with a letter or underscore and can contain alphanumerics and

underscores). They are placed just before the block and are followed by a colon, such as SOMELABEL here:

```
SOMELABEL: {
  ...statements...
  }
```

By convention, labels are all uppercase, so as not to conflict with reserved words. Labels are used with the loop control commands next, last, and redo to alter the flow of execution in your programs.

Conditionals and Loops

The if and unless statements execute blocks of code depending on whether a condition is met. These statements take the following forms:

```
if (expression) {block} else {block}

unless (expression) {block} else {block}

if (expression1) {block}
elsif (expression2) {block}
  ...
elsif (lastexpression) {block}
else {block}
```

while loops

The while statement repeatedly executes a block as long as its conditional expression is true. For example:

```
while (<INFILE>) {
    chomp;
    print OUTFILE, "$_\n";
}
```

This loop reads each line from the file opened with the filehandle INFILE and prints them to the OUTFILE filehandle. The loop will cease when it encounters an end-of-file.

If the word while is replaced by the word until, the sense of the test is reversed. The conditional is still tested before the first iteration, though.

The while statement has an optional extra block on the end called a continue block. This block is executed before every successive iteration of the loop, even if the main while block is exited early by the loop control command next. However, the continue block is not executed if the main block is exited by a last statement. The continue block is always executed before the conditional is reevaluated.

for loops

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

and the reinitialization expressions of the loop. The for loop can be defined in terms of the corresponding while loop:

```
for ($i = 1; $i < 10; $i++) {
    ...
}
```

This is the same as:

```
$i = 1;
while ($i < 10) {
    ...
}
continue {
    $i++;
}
```

foreach loops

The foreach loop iterates over a list value and sets the control variable (*var*) to be each element of the list in turn:

```
foreach var (list) {
    ...
}
```

Like the while statement, the foreach statement can also take a continue block.

Modifiers

Any simple statement may be followed by a single modifier that gives the statement a conditional or looping mechanism. This syntax provides a simpler and often more elegant method than using the corresponding compound statements. These modifiers are:

```
statement if EXPR;
statement unless EXPR;
statement while EXPR;
statement until EXPR;
```

For example:

```
$i = $num if ($num < 50); # $i will be less than 50
$j = $cnt unless ($cnt < 100); # $j will equal 100 or greater
$lines++ while <FILE>;
print "$_\n" until /The end/;
```

The conditional is evaluated first with the while and until modifiers except when applied to a do {} statement, in which case the block executes once before the conditional is evaluated. For example:

```
do {
    $line = <STDIN>;
    ...
} until $line eq ".\n";
```

For more information on do, see Chapter 5.

Loop control

You can put a label on a loop to give it a name. The loop's label identifies the loop for the loop-control commands next, last, and redo:

```
LINE: while (<SCRIPT>) {
    print;
    next LINE if /^#/;      # Discard comments
    }
```

The syntax for the loop-control commands is:

```
last label
next label
redo label
```

If the label is omitted, the loop-control command refers to the innermost enclosing loop.

The last command is like the break statement in C (as used in loops); it immediately exits the loop in question. The continue block, if any, is not executed.

The next command is like the continue statement in C; it skips the rest of the current iteration and starts the next iteration of the loop. If there is a continue block on the loop, it is always executed just before the conditional is reevaluated.

The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed.

goto

Perl supports a goto command. There are three forms: goto *label*, goto *expr*, and goto &*name*. In general, you shouldn't need to use goto, which you'll conclude if you do a search for goto in the *comp.lang.perl.misc* archives on *www.dejanews.com*.

The goto *label* form finds the statement labeled with *label* and resumes execution there. It may not be used to go inside any construct that requires initialization, such as a subroutine or a foreach loop.

The goto *expr* form expects the expression to return a label name.

The goto &*name* form substitutes a call to the named subroutine for the currently running subroutine.

Special Variables

Some variables have a predefined, special meaning in Perl. They use punctuation characters after the usual variable indicator ($, @, or %), such as $_. The explicit, long-form names are the variables' equivalents when you use the English module by including use English; at the top of your program.

Global Special Variables

The most common special variable is `$_`, which contains the default input and pattern-searching string. For example:

```
foreach ('hickory','dickory','doc') {
        print;
}
```

The first time the loop is executed, "hickory" is printed. The second time around, "dickory" is printed, and the third time, "doc" is printed. That's because in each iteration of the loop, the current string is placed in `$_` and is used by default by `print`. Here are the places where Perl will assume `$_`, even if you don't specify it:

- Various unary functions, including functions such as `ord` and `int`, as well as the all file tests (`-f`, `-d`), except for `-t`, which defaults to `STDIN`.
- Various list functions such as `print` and `unlink`.
- The pattern-matching operations `m//`, `s///`, and `tr///` when used without an `=~` operator.
- The default iterator variable in a `foreach` loop if no other variable is supplied.
- The implicit iterator variable in the `grep` and `map` functions.
- The default place to put an input record when a line-input operation's result is tested by itself as the sole criterion of a `while` test (i.e., `<`*`filehandle`*`>`). Note that outside of a `while` test, this does not happen.

The following is a complete listing of global special variables:

`$_`
`$ARG`
: The default input and pattern-searching space.

`$.`
`$INPUT_LINE_NUMBER`
`$NR`
: The current input line number of the last filehandle that was read. An explicit close on the filehandle resets the line number.

`$/`
`$INPUT_RECORD_SEPARATOR`
`$RS`
: The input record separator; newline by default. If set to the null string, it treats blank lines as delimiters.

`$,`
`$OUTPUT_FIELD_SEPARATOR`
`$OFS`
: The output field separator for the `print` operator.

`$\`
`$OUTPUT_RECORD_SEPARATOR`
`$ORS`
: The output record separator for the `print` operator.

$
$LIST_SEPARATOR
: Like $, except that it applies to list values interpolated into a double-quoted string (or similar interpreted string). Default is a space.

$;
$SUBSCRIPT_SEPARATOR
$SUBSEP
: The subscript separator for multidimensional array emulation. Default is \034.

$^L
$FORMAT_FORMFEED
: What a format outputs to perform a formfeed. Default is \f.

$:
$FORMAT_LINE_BREAK_CHARACTERS
: The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is \n".

$^A
$ACCUMULATOR
: The current value of the write accumulator for format lines.

$#
$OFMT
: Contains the output format for printed numbers (deprecated).

$?
$CHILD_ERROR
: The status returned by the last pipe close, backtick (``) command, or system operator.

$!
$OS_ERROR
$ERRNO
: If used in a numeric context, yields the current value of the errno variable, identifying the last system call error. If used in a string context, yields the corresponding system error string.

$@
$EVAL_ERROR
: The Perl syntax error message from the last eval command.

$$
$PROCESS_ID
$PID
: The pid of the Perl process running this script.

$<
$REAL_USER_ID
$UID
: The real user ID (uid) of this process.

`$>`
`$EFFECTIVE_USER_ID`
`$EUID`
: The effective uid of this process.

`$(`
`$REAL_GROUP_ID`
`$GID`
: The real group ID (gid) of this process.

`$)`
`$EFFECTIVE_GROUP_ID`
`$EGID`
: The effective gid of this process.

`$0`
`$PROGRAM_NAME`
: Contains the name of the file containing the Perl script being executed.

`$[`
: The index of the first element in an array and of the first character in a substring. Default is 0.

`$]`
`$PERL_VERSION`
: Returns the version plus patch level divided by 1,000.

`$^D`
`$DEBUGGING`
: The current value of the debugging flags.

`$^E`
`$EXTENDED_OS_ERROR`
: Extended error message on some platforms.

`$^F`
`$SYSTEM_FD_MAX`
: The maximum system file descriptor, ordinarily 2.

`$^H`
: Contains internal compiler hints enabled by certain pragmatic modules.

`$^I`
`$INPLACE_EDIT`
: The current value of the inplace-edit extension. Use `undef` to disable inplace editing.

`$^M`
: The contents of `$M` can be used as an emergency memory pool in case Perl `die`s with an out-of-memory error. Use of `$M` requires a special compilation of Perl. See the INSTALL document for more information.

`$^O`
`$OSNAME`
: Contains the name of the operating system for which the current Perl binary was compiled.

`$^P`
`$PERLDB`
: The internal flag that the debugger clears so that it doesn't debug itself.

`$^T`
`$BASETIME`
: The time at which the script began running, in seconds since the epoch.

`$^W`
`$WARNING`
: The current value of the warning switch, either true or false.

`$^X`
`$EXECUTABLE_NAME`
: The name that the Perl binary itself was executed as. As of Perl 5.8, Perl asks the operating system instead of using C's `argv[0]`.

`$ARGV`
: Contains the name of the current file when reading from `<ARGV>`.

Global Special Arrays and Hashes

`@ARGV`
: The array containing the command-line arguments intended 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.

`@F`
: The array into which the input lines are split when the *–a* command-line switch is given.

`%INC`
: The hash containing entries for the filename of each file that has been included via `do` or `require`.

`%ENV`
: The hash containing your current environment.

`%SIG`
: The hash used to set signal handlers for various signals.

Global Special Filehandles

`ARGV`
: The special filehandle that iterates over command-line filenames in `@ARGV`. Usually written as the null filehandle in `<>`.

`STDERR`
: The special filehandle for standard error in any package.

`STDIN`
: The special filehandle for standard input in any package.

STDOUT
: The special filehandle for standard output in any package.

DATA
: The special filehandle that refers to anything following the `__END__` token in the file containing the script. Or the special filehandle for anything following the `__DATA__` token in a required file, as long as you're reading data in the same package `__DATA__` was found in.

_ *(underscore)*
: The special filehandle used to cache the information from the last `stat`, `lstat`, or file test operator.

Global Special Constants

__END__
: Indicates the logical end of your program. Any following text is ignored, but may be read via the `DATA` filehandle.

__FILE__
: Represents the filename at the point in your program where it's used. Not interpolated into strings.

__LINE__
: Represents the current line number. Not interpolated into strings.

__PACKAGE__
: Represents the current package name at compile time, or undefined if there is no current package. Not interpolated into strings.

Regular Expression Special Variables

For more information on regular expressions, see the section "Regular Expressions" later in this chapter.

$*digit*
: Contains the text matched by the corresponding set of parentheses in the last pattern matched. For example, `$1` matches whatever was contained in the first set of parentheses in the previous regular expression.

$&
$MATCH
: The string matched by the last successful pattern match.

$'
$PREMATCH
: The string preceding whatever was matched by the last successful pattern match.

$'
$POSTMATCH
: The string following whatever was matched by the last successful pattern match.

$+
$LAST_PAREN_MATCH
: The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns was matched. For example:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

$^N
: The string matched by the most recently closed group. This is most useful inside `(?{...})` blocks for examining matched text. If you have multiple matches denoted by parentheses, `$^N` can be used in lieu of `$1`, `$2`, `$3`, etc., so you don't have to manually count the number of sets of parentheses that denote your matches. For example:

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

$words = "person|here";
$words =~ /(\w+)\|(\w+)/;
print $^N;   # Prints 'here'
```


Filehandle Special Variables

Most of these variables apply only when using formats. See the section "Unicode" later in this chapter.

$|
$OUTPUT_AUTOFLUSH
: If set to nonzero, forces an `fflush(3)` after every `write` or `print` on the currently selected output channel.

$%
$FORMAT_PAGE_NUMBER
: The current page number of the currently selected output channel.

$=
$FORMAT_LINES_PER_PAGE
: The current page length (printable lines) of the currently selected output channel. Default is 60.

$-
$FORMAT_LINES_LEFT
: The number of lines left on the page of the currently selected output channel.

$~
$FORMAT_NAME
: The name of the current report format for the currently selected output channel. Default is the name of the filehandle.

$^
$FORMAT_TOP_NAME
: The name of the current top-of-page format for the currently selected output channel. Default is the name of the filehandle with `_TOP` appended.

Operators

Table 4-3 lists all the Perl operators from highest to lowest precedence and indicates their associativity.

Table 4-3. Perl associativity and operators, listed by precedence

Associativity	Operators
Left	Terms and list operators (leftward)
Left	-> (method call, dereference)
Nonassociative	++ -- (autoincrement, autodecrement)
Right	** (exponentiation)
Right	! ~ \ and unary + and - (logical not, bit-not, reference, unary plus, unary minus)
Left	=~ !~ (matches, doesn't match)
Left	* / % x (multiply, divide, modulus, string replicate)
Left	+ - . (addition, subtraction, string concatenation)
Left	<< >> (left bit-shift, right bit-shift)
Nonassociative	Named unary operators and file-test operators
Nonassociative	< > <= >= lt gt le ge (less than, greater than, less than or equal to, greater than or equal to, and their string equivalents)
Nonassociative	== != <=> eq ne cmp (equal to, not equal to, signed comparison, and their string equivalents)
Left	& (bit-and)
Left	| ^ (bit-or, bit-xor)
Left	&& (logical AND)
Left	|| (logical OR)
Nonassociative	.. ... (range)
Right	?: (ternary conditional)
Right	= += -= *= and so on (assignment operators)
Left	, => (comma, arrow comma)
Nonassociative	List operators (rightward)
Right	not (logical not)
Left	and (logical and)
Left	or xor (logical or, xor)

You can clarify your expressions by using parentheses to group any part of an expression. Anything in parentheses will be evaluated as a single unit within a larger expression.

With very few exceptions, Perl operators act upon scalar values only, not upon list values.

Terms that take highest precedence in Perl include variables, quote and quotelike operators, any expression in parentheses, and any function with arguments in parentheses.

A list operator is a function that can take a list of values as its argument. List operators take highest precedence when considering what's to the left of them. They

have considerably lower precedence when looking to their right, which is the expected result.

Also parsed as high-precedence terms are the `do{}` and `eval{}` constructs, as well as subroutine and method calls, the anonymous array and hash composers (`[]` and `{}`), and the anonymous subroutine composer `sub{}`.

A unary operator is a function that takes a single scalar value as its argument. Unary operators have a lower precedence than list operators because they expect and take only one value.

The Arrow Operator

The arrow operator is a dereference operator. It can be used for references to arrays, hashes, code references, or for calling methods on objects. See the discussion of references in Chapter 7.

Unary Operators

Unary `!` performs logical negation, that is, "not." The `not` operator is a lower-precedence version of `!`.

Unary - performs arithmetic negation if the operand is numeric. If the operand is an identifier, then a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned.

Unary ~ performs bitwise negation, that is, one's complement. For example, on a 32-bit machine, `~0xFF` is `0xFFFFFF00`. If the argument to ~ is a string instead of a number, a string of identical length is returned, but with all the bits of the string complemented.

Unary + has no semantic effect whatsoever, even on strings. It is syntactically useful for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments.

Unary `\` creates a reference to whatever follows it (see "References and Complex Data Structures" later in this chapter). Do not confuse this behavior with the behavior of the backslash within a string. The `\` operator may also be used on a parenthesized list value in a list context, in which case it returns references to each element of the list.

Arithmetic Operators

Binary ** is the exponentiation operator. Note that it binds even more tightly than unary minus, so `-2**4` is `-(2**4)`, not `(-2)**4`. Also note that ** has right associativity, so:

```
$e = 2 ** 3 ** 4;
```

evaluates to 2 to the 81st power, not 8 to the 4th power.

The * (multiply) and / (divide) operators work exactly as you might expect, multiplying or dividing their two operands. Division is done in floating-point mode, unless integer mode in enabled (via `use integer`).

The % (modulus) operator converts its operands to integers before finding the remainder according to integer division. For the same operation in floating-point mode, you may prefer to use the `fmod( )` function from the POSIX module (see Chapter 8).

Comparison Operators

Comparison operators can be categorized into relational and equality operators.

Relational operators

Perl has two classes of relational operators. One class operates on numeric values, and the other operates on string values. String comparisons are based on the ASCII collating sequence. Relational operators are nonassociative, so `$a < $b < $c` is a syntax error.

Numeric	String	Meaning
>	gt	Greater than
>=	ge	Greater than or equal to
<	lt	Less than
<=	le	Less than or equal to

Equality operators

The equal and not-equal operators return `1` for true and "" for false (just as the relational operators do). The `<=>` and *cmp* operators return `-1` if the left operand is less than the right operand, `0` if they are equal, and `+1` if the left operand is greater than the right.

Numeric	String	Meaning
==	eq	Equal to
!=	ne	Not equal to
<=>	cmp	Comparison, with signed result

Autoincrement and Autodecrement

If placed before a variable, the ++ and -- operators increment or decrement the variable before returning the value. If placed after, they increment or decrement the variable after returning the value.

Assignment Operators

Perl recognizes the following operators for assigning a value to a variable:

```
=    **=    +=    *=    &=    <<=    &&=
-=    /=    |=    >>=   ||=    .=     %=
^=    x=
```

Each operator requires a variable on the left side and an expression on the right side. For the simple assignment operator, `=`, the value of the expression is stored in the designated variable. For the other operators, Perl evaluates the expression:

```
$var OP= $value
```

as if it was written:

```
$var = $var OP $value
```

except that `$var` is evaluated only once. For example:

```
$a += 2;    # Same as $a = $a + 2
```

Pattern Match Operators

Binary `=~` binds a scalar expression to a pattern match, substitution, or translation. These operations search or modify the string `$_` by default.

Binary `!~` is just like `=~` except the return value is negated in the logical sense. The following expressions are functionally equivalent:

```
$string !~ /pattern/
not $string =~ /pattern/
```

See the section "Regular Expressions" later in this chapter.

File Test Operators

A file test operator is a unary operator that tests a filename or a filehandle.

Operator	Meaning
`-r`	File is readable by effective uid/gid
`-w`	File is writable by effective uid/gid
`-x`	File is executable by effective uid/gid
`-o`	File is owned by effective uid
`-R`	File is readable by real uid/gid
`-W`	File is writable by real uid/gid
`-X`	File is executable by real uid/gid
`-O`	File is owned by real uid
`-e`	File exists
`-z`	File has zero size
`-s`	File has nonzero size (returns size)
`-f`	File is a plain file
`-d`	File is a directory

Operator	Meaning
-l	File is a symbolic link
-p	File is a named pipe (FIFO)
-S	File is a socket
-b	File is a block special file
-c	File is a character special file
-t	Filehandle is opened to a tty
-u	File has setuid bit set
-g	File has setgid bit set
-k	File has sticky bit set
-T	File is a text file
-B	File is a binary file (opposite of -T)
-M	Age of file (at startup) in days since modification
-A	Age of file (at startup) in days since last access
-C	Age of file (at startup) in days since inode change

Logical Operators

Perl provides the && (logical AND) and || (logical OR) operators. They evaluate from left to right testing the truth of the statement.

Example	Name	Result
$a && $b	And	$a if $a is false, $b otherwise
$a || $b	Or	$a if $a is true, $b otherwise

For example, an oft-appearing idiom in Perl programs is:

```
open(FILE, "somefile") || die "Cannot open somefile: $!\n";
```

In this case, Perl first evaluates the open function. If the value is true (because somefile was successfully opened), the execution of the die function is unnecessary and is skipped.

Perl also provides lower-precedence and and or operators that are more readable.

Bitwise Operators

Perl has bitwise AND, OR, and XOR (exclusive OR) operators: &, |, and ^. These operators work differently on numeric values than they do on strings. If either operand is a number, then both operands are converted to integers, and the bitwise operation is performed between the two integers. If both operands are strings, these operators do bitwise operations between corresponding bits from the two strings.

Miscellaneous Operators

The following operators don't fit into any of the previous categories.

Range operator

The `..` range operator is really two different operators depending on the context. In a list context, it returns a list of values counting (by ones) from the left value to the right value.

In a scalar context, `..` returns a Boolean value. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, after which the range operator becomes false again. The right operand is not evaluated while the operator is in the false state, and the left operand is not evaluated while the operator is in the true state.

The alternate version of this operator, `...`, does not test the right operand immediately when the operator becomes true; it waits until the next evaluation.

Conditional operator

Ternary ?: is the conditional operator. It works like an if-then-else statement but can safely be embedded within other operations and functions:

```
test_expr ? if_true_expr : if_false_expr
```

If the *test_expr* is true, only the *if_true_expr* is evaluated. Otherwise, only the *if_false_expr* is evaluated. Either way, the value of the evaluated expression becomes the value of the entire expression.

Comma operator

In a list context, `,` is the list argument separator and inserts both its arguments into the list. In scalar context, `,` evaluates its left argument, throws that value away, then evaluates its right argument and returns that value.

The `=>` operator is mostly just a synonym for the comma operator. It's useful for documenting arguments that come in pairs. It also forces any identifier to the left of it to be interpreted as a string.

String operator

The concatenation operator `.` is used to add strings together:

```
print 'abc' . 'def';        # Prints abcdef
print $a . $b;             # Concatenates the string values of $a and $b
```

Binary `x` is the string repetition operator. In scalar context, it returns a concatenated string consisting of the left operand repeated the number of times specified by the right operand:

```
print '-' x 80;                          # Prints row of dashes
print "\t" x ($tab/8), ' ' x ($tab%8);   # Tabs over
```

In list context, if the left operand is a list in parentheses, the `x` works as a list replicator rather than a string replicator. This is useful for initializing all the elements of an array of indeterminate length to the same value:

```
@ones = (1) x 80;          # A list of 80 1s
@ones = (5) x @ones;       # Set all elements to 5
```

Regular Expressions

Regular expressions are used several ways in Perl. They're used in conditionals to determine whether a string matches a particular pattern. They're also used to find patterns in strings and replace the match with something else.

The ordinary pattern match operator looks like `/pattern/`. It matches against the `$_` variable by default. If the pattern is found in the string, the operator returns true (`1`); if there is no match, a false value ("") is returned.

The substitution operator looks like `s/pattern/replace/`. This operator searches `$_` by default. If it finds the specified *`pattern`*, it is replaced with the string in *`replace`*. If *`pattern`* is not matched, nothing happens.

You may specify a variable other than `$_` with the `=~` binding operator (or the negated `!~` binding operator, which returns true if the pattern is not matched). For example:

```
$text =~ /sampo/;
```

Pattern-Matching Operators

The following list defines Perl's pattern-matching operators. Some of the operators have alternative "quoting" schemes and have a set of modifiers that can be placed directly after the operators to affect the match operation in some way.

`m/`*`pattern`*`/gimosxe`
: Searches a string for a pattern match. Modifiers are:

Modifier	Meaning
`g`	Match globally, i.e., find all occurrences.
`i`	Do case-insensitive pattern matching.
`m`	Treat string as multiple lines.
`o`	Compile pattern only once.
`s`	Treat string as single line.
`x`	Use extended regular expressions.

If `/` is the delimiter, then the initial `m` is optional. With `m`, you can use any pair of non-alphanumeric, non-whitespace characters as delimiters.

`?`*`pattern`*`?`
: This operator is just like the `m/`*`pattern`*`/` search, except it matches only once.

`qr/`*`pattern`*`/imosx`
: Creates a precompiled regular expression from *`pattern`*, which can be passed around in variables and interpolated into other regular expressions. The modifiers are the same as those for `m//` above.

`s/`*`pattern`*`/`*`replacement`*`/egimosx`
: Searches a string for *`pattern`* and replaces any match with the *`replacement`* text. Returns the number of substitutions made, which can be more than one

with the /g modifier. Otherwise, it returns false (0). If no string is specified via the =~ or !~ operator, the $_ variable is searched and modified. Modifiers are:

Modifier	Meaning
e	Evaluate the right side as an expression.
g	Replace globally, i.e., all occurrences.
cg	Continue search after g failed. No longer supported for s/// as of Perl 5.8.
i	Do case-insensitive pattern matching.
m	Treat string as multiple lines.
o	Compile pattern only once.
s	Treat string as single line.
x	Use extended regular expressions.

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string (the /e modifier overrides this, however).

tr/*pattern1*/*pattern2*/cds
y/*pattern1*/*pattern2*/cds

This operator scans a string character by character and replaces all occurrences of the characters found in *pattern1* with the corresponding character in *pattern2*. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the $_ string is translated. Modifiers are:

Modifier	Meaning
c	Complement *pattern1*.
d	Delete found but unreplaced characters.
s	Squash duplicate replaced characters.

Regular Expression Syntax

The simplest kind of regular expression is a literal string. More complicated patterns involve the use of *metacharacters* to describe all the different choices and variations that you want to build into a pattern. Metacharacters don't match themselves, but describe something else. The metacharacters are:

Metacharacter	Meaning
\	Escapes the character(s) immediately following it
.	Matches any single character except a newline (unless /s is used)
^	Matches at the beginning of the string (or line, if /m is used)
$	Matches at the end of the string (or line, if /m is used)
*	Matches the preceding element 0 or more times
+	Matches the preceding element 1 or more times
?	Matches the preceding element 0 or 1 times
{...}	Specifies a range of occurrences for the element preceding it

Metacharacter	Meaning
`[...]`	Matches any one of the class of characters contained within the brackets
`(...)`	Groups regular expressions
`|`	Matches either the expression preceding or following it

The `.` (single dot) is a wildcard character. When used in a regular expression, it can match any single character. The exception is the newline character (`\n`), except when you use the `/s` modifier on the pattern match operator. This modifier treats the string to be matched against as a single "long" string with embedded newlines.

The `^` and `$` metacharacters are used as anchors in a regular expression. The `^` matches the beginning of a line. This character should appear only at the beginning of an expression to match the beginning of the line. The exception to this is when the `/m` (multiline) modifier is used, in which case it will match at the beginning of the string and after every newline (except the last, if there is one). Otherwise, `^` will match itself, unescaped, anywhere in a pattern, except if it is the first character in a bracketed character class, in which case it negates the class.

Similarly, `$` will match the end of a line (just before a newline character) only if it is at the end of a pattern, unless `/m` is used, in which case it matches just before every newline and at the end of a string. You need to escape `$` to match a literal dollar sign in all cases, because if `$` isn't at the end of a pattern (or placed right before a) or]), Perl will attempt to do variable interpretation. The same holds true for the `@` sign, which Perl will interpret as an array variable start unless it is backslashed.

The *, +, and `?` metacharacters are called *quantifiers*. They specify the number of times to match something. They act on the element immediately preceding them, which could be a single character (including the `.`), a grouped expression in parentheses, or a character class. The `{...}` construct is a generalized modifier. You can put two numbers separated by a comma within the braces to specify minimum and maximum numbers that the preceding element can match.

Parentheses are used to group characters or expressions. They also have the side effect of remembering what they matched so you can recall and reuse patterns with a special group of variables.

The `|` is the alternation operator in regular expressions. It matches either what's on its left side or right side. It does not affect only single characters. For example:

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

looks for any of the four words. You should use parentheses to provide boundaries for alternation:

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

This will match either "Andy" or "Andrew".

Escaped Sequences

The following table lists the backslashed representations of characters that you can use in regular expressions:

Code	Matches
`\a`	Alarm (beep)
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\f`	Formfeed
`\e`	Escape
`\033`	Any octal ASCII value
`\x7f`	Any hexadecimal ASCII value
`\x{263a}`	A wide hexadecimal value
`\c`*x*	Control-*x*
`\N{`*name*`}`	A named character

Character Classes

The `[...]` construct is used to list a set of characters (a *character class*) of which *one* will match. Brackets are often used when capitalization is uncertain in a match:

```
/[tT]here/
```

A dash (`-`) may be used to indicate a range of characters in a character class:

```
/[a-zA-Z]/;  # Match any single letter
/[0-9]/;     # Match any single digit
```

To put a literal dash in the list you must use a backslash before it (`\-`).

By placing a `^` as the first element in the brackets, you create a negated character class, i.e., it matches any character not in the list. For example:

```
/[^A-Z]/;    # Matches any character other than an uppercase letter
```

Some common character classes have their own predefined escape sequences for your programming convenience:

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

Code	Matches
\pP	Match P-named (Unicode) property
\PP	Match non-P
\X	Match extended unicode sequence

While Perl implements lc() and uc(), which you can use for testing the proper case of words or characters, you can do the same with escape sequences:

Code	Matches
\l	Lowercase until next character
\u	Uppercase until next character
\L	Lowercase until \E
\U	Uppercase until \E
\Q	Disable pattern metacharacters until \E
\E	End case modification

These elements match any single element in (or not in) their class. A \w matches only one character of a word. Using a modifier, you can match a whole word, for example, with \w+. The abbreviated classes may also be used within brackets as elements of other character classes.

Anchors

Anchors don't match any characters; they match places within a string. The two most common anchors are ^ and $, which match the beginning and end of a line, respectively. The following table lists the anchoring patterns used to match certain boundaries in regular expressions:

Assertion	Meaning
^	Matches at the beginning of the string (or line, if /m is used)
$	Matches at the end of the string (or line, if /m is used)
\b	Matches at word boundary (between \w and \W)
\B	Matches except at word boundary
\A	Matches at the beginning of the string
\Z	Matches at the end of the string or before a newline
\z	Matches only at the end of the string
\G	Matches where previous m//g left off
\c	Suppresses resetting of search position when used with \g. Without \c, search pattern is reset to the beginning of the string.

The $ and \Z assertions can match not only at the end of the string, but also one character earlier than that, if the last character of the string is a newline.

Quantifiers

Quantifiers are used to specify the number of instances of the previous element that can match. For instance, you could say "match any number of a's, including none" (`a*`), or "match between 5 and 10 instances of the word 'owie' (`(owie){5,10}`)".

Quantifiers, by nature, are greedy. That is, the way the Perl regular expression "engine" works is that it will look for the biggest match possible (the farthest to the right) unless you tell it not to. Say you are searching a string that reads:

```
a whatever foo, b whatever foo
```

and you want to find `a` and `foo` with something in between. You might use:

```
/a.*foo/
```

A `.` followed by a `*` looks for any character, any number of times, until `foo` is found. But since Perl will look as far to the right as possible to find `foo`, the first instance of `foo` is swallowed up by the greedy `.*` expression.

Therefore, all the quantifiers have a notation that allows for minimal matching, so they are nongreedy. This notation uses a question mark immediately following the quantifier to force Perl to look for the earliest available match (farthest to the left). The following table lists the regular expression quantifiers and their nongreedy forms:

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}`)

Pattern Match Variables

Parentheses not only group elements in a regular expression, they also remember the patterns they match. Every match from a parenthesized element is saved to a special, read-only variable indicated by a number. You can recall and reuse a match by using these variables.

Within a pattern, each parenthesized element saves its match to a numbered variable, in order starting with `1`. You can recall these matches within the expression by using `\1`, `\2`, and so on.

Outside of the matching pattern, the matched variables are recalled with the usual dollar sign, i.e., `$1`, `$2`, etc. The dollar sign notation should be used in the replacement expression of a substitution and anywhere else you might want to use the variables in your program. For example, to implement "i before e, except after c":

```
s/([^c])ei/$1ie/g;
```

The backreferencing variables are:

`$+` Returns the last parenthesized pattern match

`$&` Returns the entire matched string

`$'` Returns everything before the matched string

`$´` Returns everything after the matched string

Backreferencing with these variables will slow down your program noticeably for all regular expressions.

Extended Regular Expressions

Perl defines an extended syntax for regular expressions. The syntax is a pair of parentheses with a question mark as the first thing within the parentheses. The character after the question mark gives the function of the extension. The extensions are:

`(?#`*`text`*`)`
: A comment. The text is ignored.

`(?:...)`
`(?imsx-imsx:...)`
: This groups things like `(...)` but doesn't make backreferences.

`(?=...)`
: A zero-width positive lookahead assertion. For example, `/\w+(?=\t)/` matches a word followed by a tab, without including the tab in `$&`.

`(?!...)`
: A zero-width negative lookahead assertion. For example, `/foo(?!bar)/` matches any occurrence of `foo` that isn't followed by `bar`.

`(?<...)`
: A zero-width positive lookbehind assertion. For example, `/(?<bad)boy/` matches the word `boy` that follows `bad`, without including `bad` in `$&`. This works only for fixed-width lookbehind.

`(?{`*`code`*`})`
: An experimental regular expression feature to evaluate any embedded Perl code. This evaluation always succeeds, and *`code`* is not interpolated.

`(?<!=...)`
: A zero-width negative lookbehind assertion. For example, `/(?<!=bad)boy/` matches any occurrence of `boy` that doesn't follow `bad`. This works only for fixed-width lookbehind.

`(?>...)`
: Matches the substring that the standalone pattern would match if anchored at the given position.

`(?(`*`condition`*`)`*`yes-pattern`*`|`*`no-pattern`*`)`
`(?(`*`condition`*`)`*`yes-pattern`*`)`
: Matches a pattern determined by a condition. *`condition`* should be either an integer, which is true if the pair of parentheses corresponding to the integer

has matched, or a lookahead, lookbehind, or evaluate, zero-width assertion. *no-pattern* will be used to match if the condition was not meant, but it is also optional.

`(?imsx-imsx)`
: One or more embedded pattern-match modifiers. Modifiers are switched off if they follow a - (dash). The modifiers are defined as follows:

Modifier	Meaning
`i`	Do case-insensitive pattern matching.
`m`	Treat string as multiple lines.
`s`	Treat string as single line.
`x`	Use extended regular expressions.

Subroutines

Subroutines are declared using one of these forms:

```
sub name {block}
sub name (proto) {block}
```

Prototypes allow you to put constraints on the arguments you provide to your subroutines.

You can also create anonymous subroutines at runtime, which will be available for use through a reference:

```
$subref = sub {block};
```

Calling Subroutines

The ampersand (`&`) is the identifier used to call subroutines. Most of the time, however, subroutines can be used in an expression just like built-in functions. To call subroutines directly:

```
name(args);       # & is optional with parentheses
name args;        # Parens optional if predeclared/imported
&name;            # Passes current @_ to subroutine
```

To call subroutines indirectly (by name or by reference):

```
&$subref(args);   # & is not optional on indirect call
&$subref;         # Passes current @_ to subroutine
```

Passing Arguments

All arguments to a subroutine are passed as a single, flat list of scalars, and return values are returned the same way. Any arrays or hashes passed in these lists will have their values interpolated into the flattened list.

Any arguments passed to a subroutine come in as the array `@_`.

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

Passing References

If you want to pass more than one array or hash into or out of a function and maintain their integrity, you should pass references as arguments. The simplest way to do this is to take your named variables and put a backslash in front of them in the argument list:

```
@returnlist = ref_conversion(\@temps1, \@temps2, \@temps3);
```

This sends references to the three arrays to the subroutine (and saves you the step of creating your own named references to send to the function). The references to the arrays are passed to the subroutine as the three-member `@_` array. The subroutine will have to dereference the arguments so that the data values may be used.

Returning references is a simple matter of returning scalars that are references. This way, you can return distinct hashes and arrays.

Private and Local Variables

Any variables you use in the function that aren't declared private are global variables. In subroutines, you'll often want to use variables that won't be used anywhere else in your program, and you don't want them taking up memory when the subroutine is not being executed. You also might not want to alter variables in subroutines that might have the same name as global variables.

The `my` function declares variables that are *lexically scoped* within the subroutine. Lexically scoped variables are private variables that exist only within the block or subroutine in which they are declared. Outside of their scope, they are invisible and can't be altered in any way.

To scope multiple variables at once, use a list in parentheses. You can also assign a variable in a `my` statement:

```
my @list = (44, 55, 66);
my $cd = "orb";
```

Dynamic variables are visible to other subroutines called from within their scope, are defined with `local`, and are not private variables but global variables with temporary values. When a subroutine is executed, the global value is hidden away, and the local value is used. Once the scope is exited, the original global value is used. Most of the time, you will want to use `my` to localize parameters in a subroutine.

Prototypes

Prototypes allow you to design your subroutines to take arguments with constraints on the number of parameters and types of data. To declare a function with prototypes, use the prototype symbols in the declaration line like this:

```
sub addem ($$) {
...
}
```

In this case, the function expects two scalar arguments. The following table gives the various prototype symbols:

Symbol	Meaning
$	Scalar
@	List
%	Hash
&	Anonymous subroutine
*	Typeglob

A backslash placed before one of these symbols forces the argument to be that exact variable type. For instance, a function that requires a hash variable would be declared like this:

```
sub hashfunc (\%);
```

Unbackslashed `@` or `%` symbols act exactly alike and will eat up all remaining arguments, forcing list context. Likewise, a `$` forces scalar context on an argument, so taking an array or hash variable for that parameter would probably yield unwanted results.

A semicolon separates mandatory arguments from optional arguments. For example:

```
sub newsplit (\@$;$);
```

requires two arguments: an array variable and a scalar. The third scalar is optional. Placing a semicolon before `@` and `%` is not necessary since lists can be null.

A typeglob prototype symbol (*) will always turn its argument into a reference to a symbol table entry. It is most often used for filehandles.

References and Complex Data Structures

A Perl reference is a fundamental data type that "points" to another piece of data or code. A reference knows the location of the information and the type of data stored there.

A reference is a scalar and can be used anywhere a scalar can be used. Any array element or hash value can contain a reference (a hash key cannot contain a reference), which is how nested data structures are built in Perl. You can construct lists containing references to other lists, which can contain references to hashes, and so on.

Creating References

You can create a reference to an existing variable or subroutine by prefixing it with a backslash:

```
$a = "fondue";
@alist = ("pitt", "hanks", "cage", "cruise");
```

```
%song = ("mother" => "crying", "brother" => "dying");
sub freaky_friday { s/mother/daughter/ }
# Create references
$ra = \$a;
$ralist = \@alist;
$rsong = \%song;
$rsub = \&freaky_friday; # '&' required for subroutine names
```

References to scalar constants are created similarly:

```
$pi = \3.14159;
$myname = \"Charlie";
```

Note that all references are prefixed by a `$`, even if they refer to an array or hash. All references are scalars; thus, you can copy a reference to another scalar or even reference another reference:

```
$aref = \@names;
$bref = $aref;            # Both refer to @names
$cref = \$aref;           # $cref is a reference to $aref
```

Because arrays and hashes are collections of scalars, you can create references to individual elements by prefixing their names with backslashes:

```
$star = \$alist[2];       # Refers to third element of @alist
$action = \$song{mother}; # Refers to the 'mother' value of %song
```

Referencing anonymous data

It is also possible to take references to literal data not stored in a variable. This data is called *anonymous* because it is not bound to any named variable.

To create a reference to a scalar constant, simply backslash the literal string or number.

To create a reference to an anonymous array, place the list of values in square brackets:

```
$shortbread = [ "flour", "butter", "eggs", "sugar" ];
```

This creates a reference to an array, but the array is available only through the reference `$shortbread`.

A reference to an anonymous hash uses braces around the list of elements:

```
$cast =  { host     => "Space Ghost",
           musician => "Zorak",
           director => "Moltar" };
```

Dereferencing

Dereferencing returns the value a reference points to. The general method of dereferencing uses the reference variable substituted for the regular name part of a variable. If `$r` is a reference, then `$$r`, `@$r`, or `%$r` retrieve the value that is referred to, depending on whether `$r` is pointing to a scalar, array, or hash. A reference can be used in all the places where an ordinary data type can be used.

When a reference is accidentally evaluated as a plain scalar, it returns a string that indicates the type of data it points to and the memory address of the data.

If you just want to know the type of data that is being referenced, use `ref`, which returns one of the following strings if its argument is a reference. Otherwise, it returns false.

```
SCALAR
ARRAY
HASH
CODE
GLOB
REF
```


Arrow dereferencing

References to arrays, hashes, and subroutines can be dereferenced using the `->` operator. This operator dereferences the expression to its left, which must resolve to an array or hash and accesses the element represented by the subscripted expression on its right. For example, these three statements are equivalent:

```
$$arrayref[0] = "man";
${$arrayref}[0] = "man";
$arrayref->[0] = "man";
```

The first statement dereferences `$arrayref` first and finds the first element of that array. The second uses braces to clarify this procedure. The third statement uses the arrow notation to do the same thing.

The arrow dereferencing notation can be used only to access a single scalar value. You cannot use arrow operators in expressions that return either slices or whole arrays or hashes.

Filehandles

A filehandle is the name for an I/O connection between your Perl process and the operating system. Filehandle names are like label names but use their own namespace. Like label names, the convention is to use all uppercase letters for filehandle names.

Every Perl program has three filehandles that are automatically opened: STDIN, STDOUT, and STDERR. By default, the `print` and `write` functions write to STDOUT. Additional filehandles are created using the `open` function:

```
open (DATA, "numbers.txt");
```

DATA is the new filehandle that is attached to the external file, which is now opened for reading. You can open filehandles for reading, writing, and appending to external files and devices. To open a file for writing, prefix the filename with a greater-than sign:

```
open(OUT, ">outfile");
```

To open a file for appending, prefix the filename with two greater-than signs:

```
open(LOGFILE, ">>error_log");
```

The open function returns true if the file is successfully opened, and false if it failed to open. Opening a file can fail for any number of reasons, e.g., a file does not exist, is write-protected, or you don't have permission for a file or directory. However, a filehandle that has not been successfully opened can still be read from (giving you an immediate EOF) or written to with no noticeable effects.

You should always check the result of open immediately and report an error if the operation does not succeed. The warn function can report an error to standard error if something goes wrong, and die can terminate your program and tell you what went wrong. For example:

```
open(LOGFILE, "/usr/httpd/error_log")
        || warn "Could not open /usr/httpd/error_log.\n";
open(DATA, ">/tmp/data") || die "Could not create /tmp/data\n.";
```

Once the file is opened, you can access the data using the diamond operator, <*filehandle*>. This is the line-input operator. When used on a filehandle in a scalar context, it returns a line from a filehandle as a string. Each time it is called, it returns the next line from the filehandle until it reaches the end-of-file. The operator keeps track of the line it is on in the file, unless the filehandle is closed and reopened, which resets the operator to the top-of-file.

For example, the following prints any line containing the word "secret.html" from the LOGFILE filehandle:

```
while (<LOGFILE>) {
        print "$_\n" if /secret\.html/;
}
```

In a list context, the line-input operator returns a list in which each line is an element. The empty <> operator reads from the ARGV filehandle, which reads the array of filenames from the Perl command line. If @ARGV is empty, the operator resorts to standard input.

A number of functions send output to a filehandle. The filehandle must already be opened for writing, of course. In the previous example, print wrote to the STDOUT filehandle, even though it wasn't specified. Without a filehandle, print defaults to the currently selected output filehandle, which will be STDOUT until you open and select another one in your program. See the select function (filehandle version) for more information.

If your program involves more than a couple of open filehandles, you should specify the filehandles for all of your I/O functions to be safe:

```
print LOGFILE "======= Generated report $date ======="
```

To close a filehandle, use the close function. Filehandles are also closed when the program exits.

Perl 5.8 and PerlIO

Perl 5.8 does I/O via PerlIO instead of through your system's I/O (STDIO). By implementing open( ) with PerlIO, the default behavior of open is changed to support a three-argument format. For example:

```
open($fh, '>:utf-8', $filename)
    or die("...");   # Open $filename and support utf-8
```

In this example, the filehandle is marked with `utf-8` (or `utf8` for EBCDIC users) to accept Perl's internal Unicode encoding.

The PerlIO layers are:

unix
: Low-level read/write

stdio
: Standard I/O

perlio
: Portable implementation of buffering

crlf
: Win32

Also in Perl 5.8, you are no longer required to name a filehandle in `open()` because Perl will handle the filehandles internally:

```
open($fh, ...) or ...
```

You can also use anonymous temporary files with the new form of `open()`:

```
open($fh, ">", undef) or ...
```

Pipes can also be used with a multiple-argument form of `open`. The following code is roughly equivalent to the Unix command `'ls -al'`:

```
open($fh, "-|", 'ls -al', '/users') or ...
```

Signals

Perl supports many facilities for Interprocess Communication (IPC), including signals and sockets. Regardless of which IPC form you use with Perl, you'll almost always tangle with signals.

Signals can be generated from any source, including key sequences on your keyboard, an event on your system, or from a failed program.

For quite some time, Perl has used a simple signal handling module, in which `%SIG` was populated with keys that represented the signal handlers your system supported. You'd invoke one of these handlers when Perl came across `$SIG{'whatever_signal_here'}`. In fact, even in Perl 5.8, the `%SIG` interface is the same, but the behavior of signals is entirely different. More on this shortly.

For signals to be useful, they must be trapped and dealt with. However, you shouldn't try to do too much when you trap a signal. Generally, you should only generate a warning and deal with the condition that caused the error. In a way, Perl's `%SIG` keeps you from doing too much. `%SIG` works like this:

```
$SIG{I_GOT_THIS_SIGNAL} = \&now_call_this_sub_to_handle_it;
sub now_call_this_sub_to_handle it {
    ...
}
```

If you need to know which signals Perl is aware of, do a `'kill -l'` on a Unix system, or use *Config.pm*:

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

use Config;

print "Perl knows about the following signals:\n";
print $Config{sig_name_init}, "\n";
```

Here's a simple snippet that lets you know if someone sent you a `HUP` signal:

```
$SIG{HUP} = \&hic_hup;
sub hic_hup {
     my $signal = shift;
     # Don't die for now.  Just warn us.
     warn "Somebody sent me a SIG${signal}\n";
}
```

Often, the type of signal you encounter in your Perl program will dictate how you should handle it. For example, you should assign `'IGNORE'` to the `'CHLD'` signal on many Unix platforms to reduce the possibility of creating zombie processes. Other signals, such as `KILL` and `STOP`, cannot and should not be ignored so easily. For this purpose, Perl lets you use a `local( )` statement, which lets you temporarily ignore a signal:

```
sub kill_handler {
    local $SIG{KILL} = 'IGNORE'; # Inherited by all functions here
    i_will_settle_for_denial();
}

sub i_will_settle_for_denial {
     # KILL ignored for now.  Go in peace.
}
```

Unix allows you to kill processes with negative process IDs by sending a *signal zero*. If you have a program that starts itself and several children, use `'kill -PARENT_PID'` to kill the parent and all of the children processes. But obviously, if you send a `HUP` to the parent, you don't want the parent to die. You can avoid this with Perl and signals:

```
sub be_nice_to_your_parents {
     local($SIG{HUP}) = 'IGNORE';
      kill('HUP', -$$);    # This pid and its kids
}
```

Naturally, you don't have to do anything fancy with signal handlers. You can simply die if the given `$SIG{NAME}` won't cause negative effects on your system if you mishandle it:

```
$SIG{INT} = sub {
    die "\nYou've interrupted me for one last time!\n"
};
```

Keep in mind that it's not all fun and games with Perl and signals. If you don't know how your system's C library and its signals implementation behave, or if

you haven't read the instructions before firing your BB gun, you'll shoot your eye out. We guarantee it!

As of Perl 5.8, you can probably be a little more confident in Perl's ability to handle even a bad system-specific signals implementation. In the old days, when men were men and eyeless men were everywhere, a bad signals implementation at both a system and Perl level, or a signal cropping up at the wrong time, could corrupt Perl's internal state. Thankfully, Perl 5.8 and later will postpone signal handling until it's safe to proceed. This means that while the signals interface doesn't change, even if your program catches a signal at a specific place, Perl 5.8 and later will finish whatever they are doing when the signal is encountered.

Finally, signals aren't limited to coping with the behaviors of processes. You can also trap alarm signals for Perl functions and system calls alike if your Perl code lives in an `eval( )` block. For example, let's say you want to ping a host, `your-host.your.domain`, and you want the ping process to timeout in a reasonable amount of time if `your-host.your.domain` isn't available. You can use `alarm( )` inside of `eval( )`, like so:

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

my $to_ping = 'your-host.your.domain';
my $a_secs = 5;

eval {
    local $SIG{ALRM} = sub { die "no dice for $to_ping" };
    alarm $a_secs;
    system("/usr/sbin/ping", $to_ping);
    alarm 0;
};

if ($@ and $@ !~ /no dice for $to_ping/) { die }
```

Note that you may have to wait for the ping command itself to timeout. While `alarm` occurs after `$a_secs`, you won't see the "no dice for…" message until `system( )` (or `qx( )`) is complete. This is not the case for Perl functions such as `flock( )`, `open( )`, etc.

Unicode

Unicode provides a unique number for every character, regardless of the computing platform, program, or programming language. This is particularly important because without a standard such as Unicode, computers would continue to use different encoding classes for characters, many of which would conflict if character classes were used together.

Unicode support was introduced to Perl with Perl 5.6. Although it is still not completely adherent in the Unicode spec, Unicode support has matured significantly under Perl 5.8. You can now use Unicode reliably with file I/O and with regular expressions. With regular expressions, the pattern will adapt to the data and will automatically switch to the correct Unicode character scheme.

Perl's Unicode implementation falls into the following categories:

I/O
: There is currently no way in Perl to mark data that's read from or written to a file as being of type Unicode (utf8). Future versions of Perl will support such a feature.

Regular expressions
: The determination whether to match Unicode characters is made when the pattern is compiled, based on whether the pattern contains Unicode characters and not when matching happens at runtime. This will be changed to match Unicode characters at runtime.

`use utf8`
: The utf8 module is still needed to enable a few Unicode features. The `utf8` pragma, as implemented by the utf8 module, implements tables used for Unicode support. You must load the `utf8` pragma explicitly to enable recognition of UTF-8 encoded literals and identifiers in the source text.

Byte and character semantics
: As of 5.6.0, Perl uses logically wide characters to represent strings internally. This internal representation uses the UTF-8 encoding. Future versions of Perl will work with characters rather than bytes. This was a purposeful decision made so Perl 5.6 could transition from byte semantics to character semantics in programs. Perl will make the decision to switch to character semantics if it finds that the input data has characters on which it can safely operate with UTF-8. You can disable character semantics by using the `bytes` pragma, as explained in Chapter 8. Character semantics have the following effects:

  - Strings and patterns may contain characters that have an ordinal value larger than 255.
  - Identifiers within a Perl program may contain Unicode alphanumeric characters.
  - Regular expressions match characters and not bytes.
  - Character classes in regular expressions match characters and not bytes.
  - Named Unicode properties and block ranges may be used as character classes with the `\p` and `\P` constructs.
  - `\X` matches any extended Unicode sequence.
  - `tr//` matches characters instead of bytes.
  - Case translation operators use the Unicode case translation tables when provided character input.
  - Most operators that deal with positions or lengths in a string switch to using character positions.
  - `pack( )` and `unpack( )` do not change.
  - Bit operators work on characters.
  - `scalar reverse( )` reverses characters and not bytes.

Formats

Formats are a mechanism for generating formatted reports for outputting data and are defined with the `format` keyword. The general form looks like:

```
format name =
 ...template lines...
 ...argument line...
.
```

Most of your format names will be the same as the filehandle names for which they are used. The default format for a filehandle is the one with the same name.

The format definition is like a subroutine definition. It doesn't contain immediately executed code and can therefore be placed anywhere in the file with the rest of the program; they are commonly placed near the end of the file with subroutine definitions. To output to a format, use the `write` function instead of `print`.

The template lines contain literal text and fieldholders. Fieldholders contain symbols that describe the size and positioning of the area on the line where data is output. An argument line immediately follows a template line that contains the fields to be replaced by data. The argument line is a list of variables (or expressions), separated by commas, that fill the fields in the previous line in the order they are listed.

Here's an example of a template line with two fieldholders and the argument line that follows:

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

The fieldholders are `@<<<<<<<<<<` and `@<<`, which specify left-justified text fields with 11 and 3 characters, respectively.

Most fieldholders start with `@`. The characters following `@` indicate the type of field, while the number of characters (including `@`) indicate the field width. The following fieldholder characters determine the positioning of text fields:

`<<<<` *(left angle brackets)*
: A left-justified field; if the value is shorter than the field width, it will be padded on the right with spaces.

`>>>>` *(right angle brackets)*
: A right-justified field; if the value is too short, it will be padded on the left with spaces.

`||||` *(vertical bars)*
: A centered field; if the value is too short, it will be padded on both sides with spaces, enough on each side to center the value within the field.

Another kind of fieldholder is a *fixed-precision numeric field*. This field also begins with `@`, and is followed by one or more hashmarks (`###`) with an optional dot (indicating a decimal point). For example:

```
format MONEY =
Assets: @######.## Liabilities: @######.## Net: @######.##
```

```
    $assets, $liabilities, $assets-$liabilities
    .
```

The multiline fieldholder allows you to include a value that may have many lines of information. This fieldholder is denoted by `@*` on a line by itself. The next line defines the value that will be substituted into the field, which in this case may be an expression that results in a value that contains many newlines.

Another kind of fieldholder is a *filled field*. This fieldholder allows you to create a filled paragraph, breaking the text into conveniently sized lines at word boundaries, wrapping the lines as needed. A filled field is denoted by replacing the `@` marker in a text fieldholder with a caret (`^<<<`, for example). The corresponding value for a filled field (on the following line of the format) must be a scalar variable containing text, rather than an expression that returns a scalar value. When Perl is filling the filled field, it takes the value of the variable and removes as many words as will fit in the field. Subsequent calls for the variable in a filled field will continue where the last one left off.

If the variable's contents are exhausted before the number of fields, you will simply end up with blank lines. You can suppress blank lines by placing a tilde (`~`) on the line. Any line that contains a tilde character is not output if the line would have otherwise printed blank (i.e., just whitespace). The tilde itself always prints as a blank and can be placed anywhere a space could have been placed in the line.

If the text in the variable is longer than what can be filled in the fields, output continues only until the fields run out. The shortcut to get the string to print until it ends is to use two consecutive tildes (`~~`) on a line. This causes the line to repeat automatically until the result is a completely blank line (which will be suppressed).

Default values for format parameters all relate to the format of the currently selected filehandle. The currently selected filehandle starts out as STDOUT, which makes it easy to print things on the standard output. However, you can change the currently selected filehandle with the `select` function, which takes a single filehandle (or a scalar variable containing the name of a filehandle) as an argument. Once the currently selected filehandle is changed, it affects all future operations that depend on the currently selected filehandle.

Pod

Pod is a simple, but surprisingly capable, text formatter that uses tags to tell a translator how to format the text. The tags serve several purposes:

- They tell the formatter how to lay out text on the page.
- They provide font and cross-reference information.
- They start and stop parsing of code.

The last item is indicative of one of pod's most useful features—that it can be intermixed with Perl code. While it can be difficult to force yourself to go back and write documentation for your code after the fact, with Perl you can simply intermingle the documentation with the code, and do it all at once. It also lets you use the same text as both code documentation and user documentation.

A pod translator reads a file paragraph by paragraph, ignoring text that isn't pod, and converting it to the proper format. Paragraphs are separated by blank lines (not just by newlines). The various translators recognize three kinds of paragraphs:

Command
: Commands begin with `=`, followed immediately by the command identifier:

```
=cut
```

They can also be followed by text:

```
=head2 Second-level head
```

A blank line signals the end of the command.

Text
: A paragraph consisting of a block of text, generally filled and possibly justified, depending on the translator. For example, a command such as `=head2` will probably be followed with a text paragraph:

```
=head2 Pod

Pod is a simple, but surprisingly capable, text formatter that uses
tags to tell a translator how to format the text.
```

Verbatim
: A paragraph that is to be reproduced as is, with no filling or justification. To create a verbatim paragraph, indent each line of text with at least one space:

```
    Don't fill this paragraph. It's supposed
    to look exactly like this on the page.
    There are blanks at the beginning of each line.
```


Paragraph Tags

The following paragraph tags are recognized as valid pod commands.

=back

```
=back
```

Moves left margin back to where it was before the last `=over`. Ends the innermost `=over`/`=back` block of indented text. If there are multiple levels of indent, one `=back` is needed for each level.

=begin

```
=begin format
```

Starts a block of text that will be passed directly to a particular formatter rather than being treated as pod. For example:

```
    =begin html
```

A `=begin`/`=end` block is like `=for` except that it doesn't necessarily apply to a single paragraph.

=cut `=cut`

Indicates the end of pod text. Tells the compiler that there isn't anymore pod (for now) and to start compiling again.

=end `=end`

Ends a `=begin` block. Tells the translator to treat what follows as pod again.

=for `=for` *format*

Indicates a format change, for the next paragraph only.

=head1 `=head1` *text*

text following the tag is formatted as a top-level heading. Generally all uppercase.

=head2 `=head2` *text*

text following the tag is formatted as a second-level heading.

=item `=item` *text*

Starts a list. Lists should always be inside an `over/back` block. Many translators use the value of *text* on the first `=item` to determine the type of list:

`=item *`
: Bulleted list. An asterisk (*) is commonly used for the bullet, but can be replaced with any other single character. Followed by a blank line and then the text of the bulleted item:

```
=item *

This is the text of the bullet.
```

`=item` *n*
: Numbered list. Replace *n* with `1` on the first item, `2` on the second, and so on. Pod does not automatically generate the numbers.

`=item` *text*
: Definition list. Formats *text* as the term and the following paragraph as the body of the list item. For example:

```
=item <HTML>

Indicates the beginning of an HTML file
```

The exact appearance of the output depends on the translator you use, but it will look pretty much like this:

```
<HTML>
    Indicates the beginning of an HTML file
```


=over `=over n`

Specifies the beginning of a list, in which *n* indicates the depth of the indent. For example, `=over 4` will indent four spaces. Another `=over` before a `=back` creates nested lists. The `=over` tag should be followed by at least one `=item`.

=pod `=pod`

Indicates the beginning of pod text. A translator starts to pay attention when it sees this tag, and the compiler ignores everything from there to the next `=cut`.

Interior Sequences

In addition to the paragraph tags, pod has a set of tags that apply within text, either in a paragraph or a command. These interior sequences are:

Sequence	Function
B<*text*>	Makes text bold, usually for switches and programs
C<*code*>	Literal code
E<*escape*>	Named character:
E<gt>	Literal >
E<lt>	Literal <
E<*html*>	Non-numeric HTML entity
E<*n*>	Character number *n*, usually an ASCII character
F<*file*>	Filename
I<*text*>	Italicize *text*, usually for emphasis or variables
L<*name*>	Link (cross-reference) to *name*:
L<*name*>	Manpage
L<*name*/*ident*>	Item in a manpage
L<*name*/"*sec*">	Section in another manpage
L<"*sec*">	Section in this manpage; quotes are optional
L</"*sec*">	Same as L<"*sec*">
S<*text*>	*text* has non-breaking spaces
X<*index*>	Index entry
Z<>	Zero-width character

Pod Utilities

As mentioned earlier, a number of utility programs have been written to convert files from pod to a variety of output formats. Some of the utilities are described here, particularly those that are part of the Perl distribution. Other programs are available on CPAN.

perldoc

`perldoc [options] docname`

Formats and displays Perl pod documentation. Extracts the documentation from pod format and displays it. For all options except *–f*, *docname* is the name of the manpage, module, or program containing pod to be displayed. For *–f*, it's the name of a built-in Perl function to be displayed.

Options

–f function
: Formats and displays documentation for the specified Perl function.

–h
: Displays help message.

–l
: Displays full path to the module.

–m
: Displays entire module, both code and pod text, without formatting the pod.

–t
: Displays using text formatter instead of nroff. Faster, but output is less fancy.

–u
: Unformatted. Finds and displays the document without formatting it.

–v
: Verbose. Describes search for the file, showing directories searched and where file was found.

perldoc applies switches found in the PERLDOC environment variable before those from the command line. It searches directories specified by the PERL5LIB, PERLLIB (if PERL5LIB isn't defined), and PATH environment variables.

pod2fm

`pod2fm [options] file`

Translates pod to FrameMaker format.

Options

–book [bookname]
: If set, creates FrameMaker book file. If not specified, *bookname* defaults to `perl`; filename extension is *.book* in either case.

–[no]doc
: Whether to convert a MIF-format *.doc* output file to binary FrameMaker format. Default is *–doc*.

–format type
: Which format to copy from the template document specified with the *–template* option. Type can be a comma-separated list, and *–format* can also be specified more than once. Legal types are:

Type	Description
`all`	All types (the default)
`Character`	Character formats
`Paragraph`	Paragraph formats
`Page`	Master page layouts
`Reference`	Reference page layouts
`Table`	Table formats
`Variables`	Variable definitions
`Math`	Math definitions
`Cross`	Cross-reference definitions
`Color`	Color definitions
`Conditional`	Conditional text definitions
`Break`	Preserves page breaks; controls how the other types are used
`Other`	Preserves other format changes; controls how the other types are used

–[no]index
: Whether to generate an index. Defaults to *–noindex*.

–[no]lock
: Whether to lock file as read-only so you can use hypertext marker feature. Defaults to *–nolock*.

–[no]mmlonly
: Whether to stop execution after generating the MML version of the file. Default is *–nommlonly*.

–[no]open
: Whether to try to open the book after creating it; requires the *–book* option.

–template document
: Specifies a template document for *pod2fm* copies a format for use in formatting the output. *document* is the path to the template document.

–[no]toc
: Whether to generate a table of contents. Defaults to *–notoc*.

pod2html

`pod2html [options] inputfile`

Translates files from pod to HTML format. Wrapper around the standard module Pod::Html; see Chapter 8 for the options, which are passed to Pod::Html as arguments.

pod2latex

`pod2latex inputfile`

Translates files from pod to LaTeX format. Writes output to a file with *.tex* extension.

pod2man

`pod2man [options] inputfile`

Translates pod directives in file *inputfile* to Unix manpage format. Converts pod-tagged text to nroff source that can be viewed by the *man* command, or troff for typesetting.

Options

`--center=string`
: Sets centered header to *string*. Defaults to "User Contributed Perl Documentation" unless `-\-official` flag is set, in which case it defaults to "Perl Programmers Reference Guide."

`--date=string`
: Sets left-hand footer string to `date`.

`--fixed=font`
: Specifies the fixed-width font to use for code examples.

`--lax`
: If set, ignores missing sections.

`--official`
: If set, uses default header as shown for *-\-center* above.

`--release=rel`
: Sets centered footer. Defaults to current Perl release.

`--section=manext`
: Sets manpage section for nroff .TH macro. Default is `3` (functions) if filename ends in *.pm*; otherwise `1` (user commands).

pod2text

`pod2text < input`

Translates pod to text and displays it. A wrapper around the Pod::Text module.

Options

`--help`
: Displays help information.

`--htmlroot=name`
: Sets base URL for the HTML files to *name*.

`--index`
: Generates index at top of the HTML file (default).

`--infile=`*`name`*
: Converts pod file *`name`*. Default is to take input from STDIN.

`--libpods=`*`name`*`:...:`*`name`*
: List of page names (e.g., "perlfunc") that contain linkable `=item`s.

`--outfile=`*`name`*
: Creates HTML file *`name`*. Default is to send output to STDOUT.

`--podroot=`*`name`*
: Uses *`name`* as base directory for finding library pods.

`--podpath=`*`name`*`:...:`*`name`*
: Lists *podroot* subdirectories with pod files whose HTML-converted forms can be linked to in cross-references.

`--netscape`
: Uses Netscape HTML directives when applicable.

`--noindex`
: Does not generate index at top of the HTML file.

`--nonetscape`
: Does not use Netscape HTML directives (default).

`--norecurse`
: Does not recurse into subdirectories specified in *podpath*.

`--recurse`
: Recurses into subdirectories specified in *podpath* (default).

`--title=`*`title`*
: Specifies title for the resulting HTML file.

`--verbose`
: Displays progress messages.