Perl Beginners' Site

Perl - because programming should be fun.

Resources

Platforms

- Mac OS
- UNIX/Linux
- Windows

Common Uses

- Bio-Info
- Chat Bots and Scripting (IRC, XMPP)
- Databases
- Email
- Games and Multimedia
- GUI Development
- Multitasking and Networking
- QA and Testing
- SSH/Telnet
- Sys Admin
- Text Generation
- Text Parsing
- Web Automation
- Web/CGI
- XML

Perl Topics

- Date and Time
- Debugging
- Files and Directories
- Hashes
- Modules and Packages
- References
- Regular Expressions
- Object Oriented Perl
- Optimising and Profiling
- Security
  - Code/Markup Injection
- Scoping and Variables
- Using CPAN
  - CPAN Wrappers for Creating System Packages
  - Finding Stuff on CPAN

Advocacy

- What about Perl 6?
- "Perl", and "perl", but not "PERL"
- Get a Job!
- Why Perl is Good
- Who is Using Perl?

Site Resources

"Perl for Newbies" - Part 4 - The Perl Beginners' Site

"Perl for Perl Newbies" - Part 4 ¶

Contents ¶

Licence ¶

1. CPAN Modules ¶

CPAN stands for the "Comprehensive Perl Archive Network". It is a mirrored repository of Perl code, packaged in so-called **CPAN distributions** (or formerly known as **CPAN modules**) that can be installed (along with their dependencies) by issuing one command.

This section will show, how to install a typical CPAN module, either manually or automatically.

1.1. Manual Compilation ¶

To compile a CPAN distribution manually, you first need to download it. You can start by browsing your nearest CPAN module and downloading from there. On UNIX the following command, will download version 2.31 of the XML::Parser module.

```
$ wget http://www.cpan.org/modules/by-module/XML/XML-Parser-2.31.tar.gz
```

Next unpack it using the "tar -xvf" command:

```
$ tar -xvf XML-Parser-2.31.tar.gz
```

(if you don't have GNU tar use "gunzip" and then "tar -xvf" instead. On Windows you can use WinZip.)

Now cd to its directory and type "perl Makefile.PL" and "make".

```
$ cd XML-Parser-2.31
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for XML::Parser::Expat
Writing Makefile for XML::Parser
$ make
cp Parser/Encodings/x-sjis-cp932.enc blib/lib/XML/Parser/Encodings/x-sjis-cp932.enc
cp Parser/Encodings/iso-8859-7.enc blib/lib/XML/Parser/Encodings/iso-8859-7.enc
cp Parser/Encodings/x-euc-jp-unicode.enc blib/lib/XML/Parser/Encodings/x-euc-jp-unicode.enc
cp Parser/Encodings/iso-8859-9.enc blib/lib/XML/Parser/Encodings/iso-8859-9.enc
cp Parser/Encodings/README blib/lib/XML/Parser/Encodings/README
```

```
cp Parser/Encodings/euc-kr.enc blib/lib/XML/Parser/Encodings/euc-kr.enc
cp Parser/Encodings/big5.enc blib/lib/XML/Parser/Encodings/big5.enc
cp Parser/Encodings/windows-1250.enc blib/lib/XML/Parser/Encodings/windows-1250.enc
cp Parser/Encodings/Japanese_Encodings.msg blib/lib/XM
.
.
.
```

After the wait, the module will be compiled. It is preferable to test it first, by invoking `make test`:

```
$ make test
```

Now you can install it, by becoming a super-user and typing `make install` at the command line.

```
$ su
Password:
\# make install
```

Module-Build ¶

Note that some distributions are now using **Module-Build**. If your distribution contains a `Build.PL` file, you should run the following commands instead:

```
perl Build.PL
./Build
./Build test
./Build install
```

More information can be found in **the main Module-Build document**

1.2. The -MCPAN Interface ¶

Perl has a module called CPAN with which one can install CPAN modules along with all of the modules they depend on. To invoke it type `perl -MCPAN -e shell` at the command line while being a super-user, and follow the instructions it gives you. The `install` command can be used to automatically install modules. For example: to install the XML::XSLT module, the following can be done from the CPAN prompt:

```
cpan> install XML::XSLT
```

This will in turn install "XML::Parser" and other modules it needs.

CPANPLUS.pm ¶

CPANPLUS.pm is a more modern, modular, and enhanced alternative to CPAN.pm. It is part of perl-5.10.x and above, but can be installed separately. As such, its use is more recommended.

Operating System/Distribution - Specific Ways ¶

Normally, you should look for a suitable native package of the CPAN module for your operating system or distribution. If that fails, you should consider building your own package. Consult your distribution's help channels for more information.

2. The sprintf function ¶

The sprintf built-in function can be used to translate a format string with some conversions embedded inside, and some parameters into a formatted string. Each conversion is specified by the starting character of the percent sign (%), and can have a type and several parameters that will dictate how it will be formatted in the output string. The output string is returned by sprintf.

Here's an example that illustrates its use:

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

use strict;
use warnings;

print sprintf("Hello \"%s\"! Your lucky number is %i.\n", "Nathan", 65);
```

The output is:

```
Hello "Nathan"! Your lucky number is 65.
```

As you can see, the first conversion is taken from the first argument, and the second one from the second argument. That's how sprintf works: processing the conversions from the arguments in order.

2.1. Supported Conversions ¶

Here are some of the supported conversions:

%% An actual percent sign

%c A character with the given ASCII number

%s A string

%d A signed integer, in decimal (also %i)

%o An integer in octal

%x An integer in hexadecimal. (use %X for uppercase hex)

%e A floating point number, in scientific notation.

%f A float in fixed decimal notation.

%b An integer in binary

Here are some examples:

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

use strict;
use warnings;

print sprintf("There is %i%% of alcohol in this beverage\n", 27);
print sprintf("%s%s\n", "This string", " ends here.");
print sprintf("650 in hex is 0x%x\n", 650);
print sprintf("650 in binary is 0b%b\n", 650);
print sprintf("3.2 + 1.6 == %f\n", 3.2+1.6);
```

And their output is:

```
There is 27% of alcohol in this beverage
This string ends here.
650 in hex is 0x28a
650 in binary is 0b1010001010
3.2 + 1.6 == 4.800000
```

2.2. Flags to the conversions ¶

One can put various flags between the percent sign and before the conversion character, which alter the output. Here is a list of them:

space	Prefix positive number with a space
+	Prefix positive number with a + sign
-	Left justify the output within the specified field
0	Use zeros, not spaces to right justify.
#	Prefix non-zero octal with 0, non-zero hex with "0x", and non-zero binary with "0b"

For example (taken from perldoc -f sprintf):

```
printf '<% d>', 12;   # prints "< 12>"
```

```
printf '<%+d>', 12;   # prints "<+12>"
printf '<%6s>', 12;   # prints "<    12>"
printf '<%-6s>', 12;  # prints "<12    >"
printf '<%06s>', 12;  # prints "<000012>"
printf '<%#x>', 12;   # prints "<0xc>"
```

Note that printf formats its arguments using sprintf and then prints them using print.

2.3. Width and Max Width ¶

One can apply an optional width and max width (or for floating point numbers - precision), specifier for the conversion flags. This is a number followed by an optional dot and another number.

Here are some examples:

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

use strict;
use warnings;

printf "<%10s>\n", "Hello";       # Prints <     Hello>
printf "<%-10s>\n", "Hello";      # Prints <Hello     >
printf "<%3.5s>\n", "Longstring"; # Prints <Longs>
printf "<%.2f>\n", 3.1415926535;  # Prints <3.14>
```

3. Alternate Forms for Writing Strings ¶

Perl has several alternate forms to write the various type of strings it supports. This section will cover the variations on this theme.

3.1. q{}, qq{} and Friends ¶

Customary	Generic	Meaning	Interpolates
''	q{}	Literal	No.
""	qq{}	Literal	Yes
``	qx{}	Command	Yes (unless the delimiter is '')
(none)	qw{}	Word List	No
//	m{}	Pattern Match	Yes (unless the delimiter is '')
(none)	qr{}	Declaration of a Regex Pattern	Yes (unless the delimiter is '')

(none)	s{}{}	Substitution	Yes (unless the delimiter is '')
(none)	tr{}{}	Transliteration	No

What it means, is that you can write an interpolated string as qq followed by a matching wrapping character, inside which the string can be placed. And likewise for the other strings. Here are some examples:

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

use strict;
use warnings;


my $h = q{Hello There};
print qq|$h, world!\n|;

my $t = q#Router#;
my $y = qq($h $h $h $t);
$y =~ s!Hello!Hi!;
print qq#$y\n#;

my @arr = qw{one two three};
for my $i (0 .. $#a)
{
    print "$i: $arr[$i]\n";
}
```

The output of this is:

```
Hello There, world!
Hi There Hello There Hello There Router
0: one
1: two
2: three
```

As one can see, the wrapping characters should match assuming they are a left/right pair ({ to } etc.).

3.2. Here Document ¶

In a here document, one specifies an ending string to end the string on a separate line, and between it, one can place any string he wishes. This is useful if your string contains a lot of irregular characters.

The syntax for a here document is a << followed by the string ending sequence, followed by the end of the statement. In the lines afterwards, one places the string itself followed by its ending sequence.

Here is an example:

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

use strict;
use warnings;


my $x = "Hello";
my $str = "There you go.";
my $true = "False";

print <<"END";
The value of \$x is: "$x"
The value of \$str is: "$str"
The value of true is: "$true"

Hoola

END
```

Its output is:

```
The value of $x is: "Hello"
The value of $str is: "There you go."
The value of true is: "False"

Hoola
```

Note that if the delimeters on the terminator after the << are double-quotes ("..."), then the here-document will interpolate, and if they are single-quotes ('...'), it will not.

An unquoted ending string causes the here-doc to interpolate, in case you encounter it in the wild. Note however, that in your code, you should always quote it, so people won't have to guess what you meant.

4. Executing Other Processes ¶

Perl enables one to execute other system commands while returning control to the script. There are several different ways to do this, and they would be covered here.

4.1. The system() Command ¶

The `system()` function executes a shell command, while maintaining the same standard input, standard output and environment of the invoking script. If called with one argument, it passes this argument as is to the shell, which in turn will process it for special characters. If passed an array, it will call the command in the first member with the rest of the array as command line arguments.

If you receive an arbitrary array and you fear it may contain only one argument, you can use the `system { $cmd_line[0] } @cmd_line` notation (similar to `print()`'s).

On success, system() returns 0 (not a true value) and one should make sure to throw an exception upon failure while referencing the built-in error variable `$?`.

Here are some examples, that will only work on UNIX systems.

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

use strict;
use warnings;

(system("ls -l /") == 0)
    or die "system 'ls -l /' failed - $?";

my @args = ("ls", "-l", "/");
(system(@args) == 0)
    or die "Could not ls -l / - $?";

(system("ls -l | grep ^d | wc -l") == 0)
    or die "Could not pipeline - $?";
```

4.2. Trapping Command Output with `...` ¶

The backticks (or more generally `qx{ ... }`), can be used to trap the output of a shell command. It executes the command and returns all of its output. Interpolation is used.

If assigned to a scalar, it returns the output as a complete string. If the output is assigned to an array, the array will contain the lines of the output.

Here is an example for a program that counts the number of directories in a directory that is given as an argument:

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

use strict;
use warnings;

my $dir = shift;
```

```
# Prepare $dir for placement inside a '...' argument
# A safer way would be to use String::ShellQuote
$dir =~ s!'!'\\''!g;

my $count = `ls -l '$dir' | grep ^d | wc -l`;

if ($?)
{
    die "Error returned by ls -l command is $@.";
}

if ($count !~ /(\d+)/)
{
    # Retrieve the number via the special regex variable $1
    $count = $1;
    print "There are $count directories\n";
}
else
{
    die "Wrong output."
}
```

4.3. open() for Command Execution ¶

The open command can be used for command execution. By prefixing the filename with a pipe (|), the rest of it is interpreted as a command invocation, which accepts standard input by printing to the filehandle, and is executed after the filehandle is closed. If the last character is a pipe, then the command is executed and its standard output is fed into the filehandle where it can be read using Perl's file input mechanisms.

Here are some examples:

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

use strict;
use warnings;

open my $in, "/sbin/ifconfig |";

my (@addrs);

while (my $line = <$in>)
{
    if ($line =~ /inet addr:((\d+\.)+\d)/)
    {
        push @addrs, $1;
    }
}
close($in);
```

```
print "You have the following addresses: \n", join("\n",@addrs), "\n";
```

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

use strict;
use warnings;

# Send an E-mail to myself
# Note: this is just an example - there are modules to do this on CPAN.

open MAIL, "|/usr/sbin/sendmail shlomif\@shlomifish.org";
print MAIL "To: Shlomi Fish <shlomif\@shlomifish.org>\n";
print MAIL "From: Shlomi Fish <shlomif\@shlomifish.org>\n";
print MAIL "\n";
print MAIL "Hello there, moi!\n";
close(MAIL);
```

Pipe to @args ¶

Recent versions of Perl also have a syntax that allows opening a process for input or output using its command line arguments. These are:

```
open my $print_to_process, "|-", $cmd, @args;
print {$print_to_process} ...;
```

and:

```
open my $read_from_process, "-|", $cmd, @args;
while (my $line = <$read_from_process>)
{
.
.
.
}
```

Doing something like `open my $print_to_process, "|-", "sendmail", $to_address;` is safer than doing: `open my $print_to_process, "|-", "sendmail $to_address";` Because a malicious person may put some offending shell characters in `$to_address` and end up with something like:

```
sendmail ; rm -fr $HOME
```

4.4. String::ShellQuote ¶

When invoking raw shell commands (instead of passing a list of command line arguments) one can easily cause a situation where an interpolated string given as argument will place arbitrary code in the shell. If for example we have the following qx call:

```
my $ls_output = qx/ls '$dir'/;
```

Then `$dir` may be set to "`' ; rm -fr ~ ; '`", which will make the shell delete our entire home directory.

To overcome such problems, one should make use of the String-ShellQuote module which provides functions for safely preventing shell-code injection.

5. More about || and && ¶

`||` and `&&` return the last argument that was evaluated. Thus, `||` is useful for assigning default values, like this:

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

use strict;
use warnings;


# shift by default shifts from @ARGV in the main program
my $start = shift || 1;
my $end = shift || ($start+9);

for my $i ($start .. $end)
{
    print "$i\n";
}
```


5.1. For sort() ¶

The `||` operator can be used in `sort`, in conjunction with operators such as `cmp` or `<=>` to sort according to several criteria. For example, if you wish to sort according to the last name and if this is equal according to the first name as well, you can write the following:

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

use strict;
use warnings;


my @array =
(
    { 'first' => "Amanda", 'last' => "Smith", },
    { 'first' => "Jane", 'last' => "Arden",},
    { 'first' => "Tony", 'last' => "Hoffer", },
    { 'first' => "Shlomi", 'last' => "Fish", },
    { 'first' => "Chip", 'last' => "Fish", },
    { 'first' => "John", 'last' => "Smith", },
    { 'first' => "Peter", 'last' => "Torry", },
    { 'first' => "Michael", 'last' => "Hoffer", },
    { 'first' => "Ben", 'last' => "Smith", },
);

my @sorted_array =
    (sort
        {
            ($a->{'last'} cmp $b->{'last'}) ||
            ($a->{'first'} cmp $b->{'first'})
        }
        @array
    );

foreach my $record (@sorted_array)
{
    print $record->{'last'} . ", " . $record->{'first'} . "\n";
}
```

Its output is:

```
Arden, Jane
Fish, Chip
Fish, Shlomi
Hoffer, Michael
Hoffer, Tony
Smith, Amanda
Smith, Ben
Smith, John
Torry, Peter
```

5.2. The "and" and "or" Operators ¶

Perl supplies two operators `and` and `or` which are equivalent to `&&` and `||` except that they have a very low precedence. (lower than any other operator in fact). There's also `not` which is the ultra-low precedence equivalent of `!`.

You can use them after a statement to write error handlers.

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

use strict;
use warnings;

# Terminate if we cannot open a file.
open 0, ">", "/hello.txt" or die "Cannot open file!";

print 0 "Hello World!\n";

close(0);
```

6. Exceptions ¶

Exceptions are a mechanism to raise an error in a program which will propagate to outside blocks, until it is caught by an explicit catching block, or it terminates the program. It is a convenient way to manage errors.

In Perl, there is a statement that throws an exception, and one that catches it. The exception may escape out of function calls as well.

6.1. die and eval ¶

The statement `die` throws an exception which can be any Perl scalar. The statement `eval { ... }` catches an excpetion that was given inside it, and after it sets the special variable `$@` to be the value of the exception or undef if none was caught.

Here's an example:

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

use strict;
use warnings;


sub read_text
```

```
{
    my $filename = "../hello/there.txt" ;
    open I, "<$filename"
        or die "Could not open $filename";
    my $text = join("",<I>);
    close(I);

    return $text;
}

sub write_text
{
    my $text = shift;
    my $filename = "../there/hello.txt";
    open O, ">$filename"
        or die "Could not open $filename for writing";
    print O $text;
    close(O);
}

sub read_and_write
{
    my $text = read_text();

    write_text($text);
}

sub perform_transaction
{
    eval {
    read_and_write();
    };
    if ($@)
    {
        print "Could not perform the transaction. Reason is:\n$@\n";
    }
}

perform_transaction();
```

6.2. The Carp module ¶

The Carp module warns or throws errors with a more useful information that the normal Perl behaviour. It supplies several such methods. For more information run perldoc Carp.

6.3. The Error.pm module ¶

The Error.pm module, which is available from CPAN, supplies object oriented exception handling. Namely, one can catch exceptions of a certain class explicitly, and differentiate between several types of exceptions.

Error.pm provides a lot of syntactic sugar that tends to break easily. As such, its use is not too recommended.

On the other side, there's the Exception-Class module which provides object-oriented exceptions with no special syntactic sugar, and which works very well. Its use is highly recommended.

Throwing objects which are associated with classes is a good way to be able to handle one's exceptions programatically .

7. More System Functions ¶

Perl supplies the user with many functions useful for performing system tasks. This section will cover some of them for your own reference.

7.1. Directory Input Routines ¶

The `opendir DIRHANDLE, EXPR` function can be used to open the directory `EXPR` for reading its file and sub-directory entries. Afterwards `readdir(DIRHANDLE)` can be used to read one entry from there, or all the entries if used in list context.

Use `closedir()` to close an opened directory.

Here's an example that counts the number of mp3s in a directory:

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

use strict;
use warnings;


sub get_dir_files
{
    my $dir_path = shift;

    opendir D, $dir_path
        or die "Cannot open the directory $dir_path";

    my @entries;
    @entries = readdir(D);
```

```
    closedir(D);

    return \@entries;
}

my $dir_path = shift || ".";

my $entries = get_dir_files($dir_path);
my @mp3s = (grep { /\.mp3$/ } @$entries);

print "You have " . scalar(@mp3s) . " mp3s in $dir_path.\n";
```

7.2. Random File I/O ¶

Perl provides mechanisms for moving to certain positions in files, and reading blocks of a certain size.

`seek FILEHANDLE, POSITION, WHENCE` sets the filehandle position within the file in bytes. If you specify `use Fcntl;` at the beginning of your program, then WHENCE can be `SEEK_SET` for start of file, `SEEK_CUR` for the current position and `SEEK_END` for the end of file.

`tell FILEHANDLE` returns the position of the current file cursor in bytes from the beginning of the file.

`read FILEHANDLE, SCALAR, LENGTH` reads `LENGTH` characters from `FILEHANDLE` into the `SCALAR` variable.

Here's an example that replaces bytes 64-127 in a file with their rot13 equivalent:

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

use strict;
use warnings;

use Fcntl;

my $filename = shift;

open F, "+<$filename"
    or die "Could not open file";

# Read bytes 64-127 into $text
seek(F, 64, SEEK_SET);

my $text;
read(F,$text,64);
# Do the actual rot13'ing with the tr command
$text =~ tr/A-Za-z/N-ZA-Mn-za-m/;
# Write them at position 64
seek(F, 64, SEEK_SET);
```

```
print F $text
close(F);
```

7.3. File Tests (-e, -d...) ¶

Perl provides several file tests that can be used to test a filehandle or filename for various conditions. `-e filename` determines if `filename` exists. `-r` determines if the file is readable, `-w` if it's writeable, `-x` if it is executable and so on.

`-d` determines if the file is a directory, and `-f` if it's a plain file. For a list of other file tests and their use consult `perldoc -f -X`.

7.4. chdir(), getcwd() and mkdir(). ¶

The built-in function `chdir EXPR` can be used to change the working directory of the program to a new value. If `EXPR` is omitted it changes to the home directory.

By `use`'ing the Cwd module, one can invoke the `getcwd()` function that will retrieve the current working directory. This is similar to the `pwd` command on UNIX shells.

Finally, `mkdir FILENAME, MASK` can be used to create a new directory with the permissions mask of `MASK`

7.5. The stat() Function ¶

The `stat` function can be used to retrieve a 13-element array that gives status information for a given file or filehandle.

The syntax is:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
    $atime,$mtime,$ctime,$blksize,$blocks)
        = stat($filename);
```

Here, `$mode` is the file mode, `$size` is the size of the file, `$atime` is the last access time (in seconds since the epoch), `$mtime` is the last modification time.

For more information about `stat` consult `perldoc -f stat` or its entry in `perlfunc`.

Webmaster: Shlomi Fish (Email - shlomif@shlomifish.org)

Original Design: GoFlexiblePro | Author: G. Wolfgang | W3C XHTML5 | W3C CSS 3

Hosted by: Hexten.net.