

NAME

`perlfm` - Perl formats

DESCRIPTION

Perl has a mechanism to help you generate simple reports and charts. To facilitate this, Perl helps you code up your output page close to how it will look when it's printed. It can keep track of things like how many lines are on a page, what page you're on, when to print page headers, etc. Keywords are borrowed from FORTRAN: `format()` to declare and `write()` to execute; see their entries in *perlfunc*. Fortunately, the layout is much more legible, more like BASIC's PRINT USING statement. Think of it as a poor man's `nroff(1)`.

Formats, like packages and subroutines, are declared rather than executed, so they may occur at any point in your program. (Usually it's best to keep them all together though.) They have their own namespace apart from all the other "types" in Perl. This means that if you have a function named "Foo", it is not the same thing as having a format named "Foo". However, the default name for the format associated with a given filehandle is the same as the name of the filehandle. Thus, the default format for STDOUT is named "STDOUT", and the default format for filehandle TEMP is named "TEMP". They just look the same. They aren't.

Output record formats are declared as follows:

```
format NAME =  
FORMLIST  
.
```

If the name is omitted, format "STDOUT" is defined. A single "." in column 1 is used to terminate a format. FORMLIST consists of a sequence of lines, each of which may be one of three types:

1. A comment, indicated by putting a '#' in the first column.
2. A "picture" line giving the format for one output line.
3. An argument line supplying values to plug into the previous picture line.

Picture lines contain output field definitions, intermingled with literal text. These lines do not undergo any kind of variable interpolation. Field definitions are made up from a set of characters, for starting and extending a field to its desired width. This is the complete set of characters for field definitions:

```
@    start of regular field  
^    start of special field  
<    pad character for left adjustification  
|    pad character for centering  
>    pad character for right adjustification  
#    pad character for a right justified numeric field  
0    instead of first #: pad number with leading zeroes  
.    decimal point within a numeric field  
...  terminate a text field, show "..." as truncation evidence  
@*   variable width field for a multi-line value  
^*   variable width field for next line of a multi-line value  
~    suppress line with all fields empty  
~~   repeat line until all fields are exhausted
```

Each field in a picture line starts with either "@" (at) or "^" (caret), indicating what we'll call, respectively, a "regular" or "special" field. The choice of pad characters determines whether a field is textual or numeric. The tilde operators are not part of a field. Let's look at the various possibilities in detail.

Text Fields

The length of the field is supplied by padding out the field with multiple "<", ">", or "|" characters to specify a non-numeric field with, respectively, left justification, right justification, or centering. For a regular field, the value (up to the first newline) is taken and printed according to the selected justification, truncating excess characters. If you terminate a text field with "...", three dots will be shown if the value is truncated. A special text field may be used to do rudimentary multi-line text block filling; see *Using Fill Mode* for details.

Example:

```
format STDOUT =
@<<<<<<  @| | | | | |  @>>>>>>
"left",    "middle",  "right"
```

Output:

```
left      middle      right
```

Numeric Fields

Using "#" as a padding character specifies a numeric field, with right justification. An optional "." defines the position of the decimal point. With a "0" (zero) instead of the first "#", the formatted number will be padded with leading zeroes if necessary. A special numeric field is blanked out if the value is undefined. If the resulting value would exceed the width specified the field is filled with "#" as overflow evidence.

Example:

```
format STDOUT =
@###  @.###  @##.###  @###  @###  ^#####
42,    3.1415,  undef,    0, 10000,  undef
```

Output:

```
42    3.142    0.000    0    #####
```

The Field @* for Variable Width Multi-Line Text

The field "@*" can be used for printing multi-line, nontruncated values; it should (but need not) appear by itself on a line. A final line feed is chopped off, but all other characters are emitted verbatim.

The Field ^* for Variable Width One-line-at-a-time Text

Like "@*", this is a variable width field. The value supplied must be a scalar variable. Perl puts the first line (up to the first "\n") of the text into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. The variable will *not* be restored.

Example:

```
$text = "line 1\nline 2\nline 3";
format STDOUT =
Text: ^*
      $text
~~    ^*
      $text
```

Output:

```
Text: line 1
      line 2
      line 3
```


[illegible]

It is possible to intermix print()s with write()s on the same output channel, but you'll have to handle \$- (\$FORMAT LINES LEFT) yourself.

Format Variables

The current format name is stored in the variable `$~ ($FORMAT_NAME)`, and the current top of form format name is in `$^ ($FORMAT_TOP_NAME)`. The current output page number is stored in `$% ($FORMAT_PAGE_NUMBER)`, and the number of lines on the page is in `$= ($FORMAT_LINES_PER_PAGE)`. Whether to autoflush output on this handle is stored in `$| ($OUTPUT_AUTOFLUSH)`. The string output before each top of page (except the first) is stored in `$^L ($FORMAT_FORMFEED)`. These variables are set on a per-filehandle basis, so you'll need to `select()` into a different one to affect them:

```
select((select(OUTF),
  $~ = "My_Other_Format",
  $^ = "My_Top_Format"
)[0]);
```

Pretty ugly, eh? It's a common idiom though, so don't be too surprised when you see it. You can at least use a temporary variable to hold the previous filehandle: (this is a much better approach in general, because not only does legibility improve, you now have intermediary stage in the expression to single-step the debugger through):

```
$ofh = select(OUTF);
$~ = "My_Other_Format";
$^ = "My_Top_Format";
select($ofh);
```



```
$string = swrite(<<'END', 1, 2, 3);
Check me out
@<<< @||| @>>>
END
print $string;
```

WARNINGS

The lone dot that ends a format can also prematurely end a mail message passing through a misconfigured Internet mailer (and based on experience, such misconfiguration is the rule, not the exception). So when sending format code through mail, you should indent it so that the format-ending dot is not on the left margin; this will prevent SMTP cutoff.

Lexical variables (declared with "my") are not visible within a format unless the format is declared within the scope of the lexical variable. (They weren't visible at all before version 5.001.)

Formats are the only part of Perl that unconditionally use information from a program's locale; if a program's environment specifies an LC_NUMERIC locale, it is always used to specify the decimal point character in formatted output. Perl ignores all other aspects of locale handling unless the `use locale` pragma is in effect. Formatted output cannot be controlled by `use locale` because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure. See *perllocale* for further discussion of locale handling.

Within strings that are to be displayed in a fixed length text field, each control character is substituted by a space. (But remember the special meaning of `\r` when using fill mode.) This is done to avoid misalignment when control characters "disappear" on some output media.