NAME

perlform - 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 justification
|    pad character for centering
>    pad character for right justification
#    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 chomped 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
```

Specifying Values

The values are specified on the following format line in the same order as the picture fields. The expressions providing the values must be separated by commas. They are all evaluated in a list context before the line is processed, so a single list expression could produce multiple list elements. The expressions may be spread out to more than one line if enclosed in braces. If so, the opening brace must be the first token on the first line. If an expression evaluates to a number with a decimal part, and if the corresponding picture specifies that the decimal part should appear in the output (that is, any picture except multiple "#" characters **without** an embedded "."), the character used for the decimal point is determined by the current LC_NUMERIC locale if "use locale" is in effect. This means that,

if, for example, the run-time environment happens to specify a German locale, "," will be used instead of the default ".". See perllocale and "WARNINGS" for more information.

Using Fill Mode

On text fields the caret enables a kind of fill mode. Instead of an arbitrary expression, the value supplied must be a scalar variable that contains a text string. Perl puts the next portion 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. (Yes, this means that the variable itself is altered during execution of the **write()** call, and is not restored.) The next portion of text is determined by a crude line-breaking algorithm. You may use the carriage return character ("\r") to force a line break. You can change which characters are legal to break on by changing the variable $: (that's $FORMAT_LINE_BREAK_CHARACTERS if you're using the English module) to a list of the desired characters.

Normally you would use a sequence of fields in a vertical stack associated with the same scalar variable to print out a block of text. You might wish to end the final field with the text "...", which will appear in the output if the text was too long to appear in its entirety.

Suppressing Lines Where All Fields Are Void

Using caret fields can produce lines where all fields are blank. You can suppress such lines by putting a "~" (tilde) character anywhere in the line. The tilde will be translated to a space upon output.

Repeating Format Lines

If you put two contiguous tilde characters "~~" anywhere into a line, the line will be repeated until all the fields on the line are exhausted, i.e. undefined. For special (caret) text fields this will occur sooner or later, but if you use a text field of the at variety, the expression you supply had better not give the same value every time forever! ("shift(@f)" is a simple example that would work.) Don't use a regular (at) numeric field in such lines, because it will never go blank.

Top of Form Processing

Top-of-form processing is by default handled by a format with the same name as the current filehandle with "_TOP" concatenated to it. It's triggered at the top of each page. See "write" in perlfunc.

Examples:

```
# a report on the /etc/passwd file
format STDOUT_TOP =
                 Passwd File
Name             Login    Office   Uid   Gid Home
------------------------------------------------------------------
```

```
.
format STDOUT =
@<<<<<<<<<<<<<<<<<<< @||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
$name,              $login,  $office,$uid,$gid, $home
.


# a report from a bug report form
format STDOUT_TOP =
                    Bug Reports
@<<<<<<<<<<<<<<<<<<<<<<<     @|||         @>>>>>>>>>>>>>>>>>>>>>>>>
$system,                     $%,          $date
------------------------------------------------------------------
.
format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $subject
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $index,                       $description
Priority: @<<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $priority,       $date,   $description
From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $from,                        $description
Assigned to: @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
             $programmer,           $description
~                          ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                           $description
~                          ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                           $description
~                          ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                           $description
~                          ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                           $description
~                          ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<...
                           $description
.
```

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 an intermediary stage in the expression to single-step the debugger through):

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

If you use the English module, you can even read the variable names:

```
use English;
$ofh = select(OUTF);
$FORMAT_NAME     = "My_Other_Format";
$FORMAT_TOP_NAME = "My_Top_Format";
select($ofh);
```

But you still have those funny **select()**s. So just use the FileHandle module. Now, you can access these special variables using lowercase method names instead:

```
use FileHandle;
format_name     OUTF "My_Other_Format";
format_top_name OUTF "My_Top_Format";
```

Much better!

NOTES

Because the values line may contain arbitrary expressions (for at fields, not caret fields), you can farm out more sophisticated processing to other functions, like **sprintf()** or one of your own. For example:

```
format Ident =
   @<<<<<<<<<<<<<<<
   &commify($n)
.
```

To get a real at or caret into the field, do this:

```
format Ident =
I have an @ here.
      "@"
.
```

To center a whole line of text, do something like this:

```
format Ident =
@|||||||||||||||||||||||||||||||||||||||||||||||||
      "Some text line"
.
```

There is no builtin way to say "float this to the right hand side of the page, however wide it is." You have to specify where it goes. The truly desperate can generate their own format on the fly, based on the current number of columns, and then **eval()** it:

```
$format  = "format STDOUT = \n"
      . '^' . '<' x $cols . "\n"
      . '$entry' . "\n"
      . "\t^" . "<" x ($cols-8) . "~~\n"
      . '$entry' . "\n"
      . ".\n";
print $format if $Debugging;
eval $format;
die $@ if $@;
```

Which would generate a format looking something like this:

```
 format STDOUT =
```

```
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$entry
      ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
$entry
.
```

Here's a little program that's somewhat like **fmt**(1):

```
format =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
$_

.

$/ = '';
while (<>) {
   s/\s*\n\s*/ /g;
   write;
}
```

Footers

While $FORMAT_TOP_NAME contains the name of the current header format, there is no corresponding mechanism to automatically do the same thing for a footer. Not knowing how big a format is going to be until you evaluate it is one of the major problems. It's on the TODO list.

Here's one strategy: If you have a fixed-size footer, you can get footers by checking $FORMAT_LINES_LEFT before each **write()** and print the footer yourself if necessary.

Here's another strategy: Open a pipe to yourself, using "open(MYSELF, "|-")" (see "open" in perlfunc) and always **write()** to MYSELF instead of STDOUT. Have your child process massage its STDIN to rearrange headers and footers however you like. Not very convenient, but doable.

Accessing Formatting Internals

For low-level access to the formatting mechanism, you may use **formline()** and access $^A (the $ACCUMULATOR variable) directly.

For example:

```
$str = formline <<'END', 1,2,3;
@<<<  @|||  @>>>
```

```
    END

    print "Wow, I just stored '$^A' in the accumulator!\n";
```

Or to make an **swrite()** subroutine, which is to **write()** what **sprintf()** is to **printf()**, do this:

```
    use Carp;
    sub swrite {
        croak "usage: swrite PICTURE ARGS" unless @_;
        my $format = shift;
        $^A = "";
        formline($format,@_);
        return $^A;
    }

    $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.

If a program's environment specifies an LC_NUMERIC locale and "use locale" is in effect when the format is declared, the locale is used to specify the decimal point character in formatted output. Formatted output cannot be controlled by "use locale" at the time when **write()** is called. 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.