Perl Best Practices

Reference Guide

Code Layout
1. Brace and parenthesis in K&R style.
2. Separate your control keywords from the following opening bracket.
3. Don’t separate subroutine or variable names from the following opening bracket.
4. Separate complex keys or indices from their surrounding brackets.
5. Use whitespace to help binary operators stand out from their operands.
6. Place a semicolon after every statement.
7. Place a comma after every value in a multiline list.
8. Use 76-column lines.
9. Use four-column indentation levels.
10. Indent with spaces, not tabs.
11. Never place two statements on the same line.
13. In paragraphs.
14. Don’t cuddle an else.
15. Align corresponding items vertically.
16. Break long expressions before an operator.
17. Factor out long expressions in the middle of statements.
18. Always break a long expression at the operator of the lowest possible precedence.
20. Format cascaded ternary operators in columns.
22. Enforce your chosen layout style mechanically.

Naming Conventions
23. Use grammatical templates when forming identifiers.
24. Name booleans after their associated test.
25. Mark variables that store references with a _ref suffix.
26. Name arrays in the plural and hashes in the singular.
27. Use underscores to separate words in multiword identifiers.
28. Distinguish different program components by case.
29. Abbr idents by prefx.
30. Abbreviate only when the meaning remains unambiguous.
31. Avoid using inherently ambiguous words in names.
32. Prefix for internal use only’ subroutines with an underscore.

Values and Expressions
33. Use interpolating string delimiters only for strings that actually interp.
34. Don’t use ‘‘ or ‘‘ for an empty string.
35. Don’t write one-character strings in visually ambiguous ways.
36. Use named character escapes instead of numeric escapes.
37. Use named constants, but don’t use constant.
38. Don’t pad decimal numbers with leading zeros.
39. Use underscores to improve the readability of long numbers.
40. Lay out multiline strings over multiple lines.
41. Use a heredoc when a multiline string exceeds two lines.
42. Use a ‘heredoc’ when a heredoc would compromise your indenta.
43. Make every heredoc terminator a single uppercase identifier with a standard prefix.
44. When introducing a heredoc, quote the terminator.
45. Don’t use barewords.
46. Reserve for pairs.
47. Don’t use commas to sequence statements.
48. Don’t mix high- and low-precedence booleans.
49. Parenthesize every raw list.
50. Use table lookup to test in membership of lists of strings, use any() for membership of lists of anything else.

Variables
51. Avoid using non-lexical variables.
52. Don’t use packages in your own development.
53. If you’re forced to modify a package variable, localize it.
54. Initialize any variable you localize.
55. Use English for the less familiar punctuation variables.
56. If you’re forced to modify a punctuation variable, localize it.
57. Don’t use the regex match variables.
58. Beware of any modification via $_.
59. Use negative indices when counting from the end of an array.
60. Take advantage of hash and array slicing.
61. Use a tabular layout for slices.
62. Factor large key or index lists out of their slices.

Control Structures
63. Use block if, not postfix if.
64. Reserve postfix if for flow-of-control statements.
65. Don’t use postfix unless for, while, or until.
66. Don’t use unless or until at all.
67. Avoid G-style statements.
68. Avoid subscripting arrays or hashes within loops.
69. Never subscript more than once in a loop.
70. Use named lexicals as explicit for loop iterators.
71. Always declare a for loop iterator variable with my.
72. Use map instead of for when generating new lists from old.
73. Use grep and instead of for when searching for values in a list.
74. Use instead of map when transforming a list in place.
75. Use a subroutine call to factor out complex list transformations.
76. Never modify $_ in a list function.
77. Avoid cascading an if.
78. Use table look-up in preference to cascaded equality tests.
79. When producing a value, use tabular ternaries.
80. Don’t use do...while loops.
81. Reject as many iterations as possible, as early as possible.
82. Don’t contort loop structures just to consolidate control.
83. Use for instead of for instead of an irregularly counted while.
84. Label every loop that is exited explicitly, and use the label with every next, last, or redo.

Built-in Functions
85. Distinguish user documentation from technical documentation.
86. Create standard POD templates for modules and applications.
87. Extend and customize your standard POD templates.
88. Put user documentation in source files.
89. Keep all user documentation in a single place within your source file.
90. Place POD as close as possible to the end of the file.
91. Subdivide your technical documentation appropriately.
92. Use block templates for major comments.
93. Use full-line comments to explain the algorithm.
94. Use end-of-line comments to point out subtleties and oddities.
95. Comment anything that has puzzled or tricked you.
96. Consider whether it’s better to rewire than to comment.
97. Use ‘invisible’ POD sections for longer technical discussions.
98. Check the spelling, syntax, and sanity of your documentation.

Documentation
99. Don’t recompute sort keys inside a sort.
100. Use reverse to reverse a list.
101. Use scalar reverse to reverse a scalar.
102. Use unpack to extract fixed-width fields.
103. Use split to extract simple variable-width fields.
104. Use Text::CSV_XS to extract complex variable-width fields.
105. Avoid scalar eval.
106. Consider building your sorting routines with Sort::Maker.
107. Use 4-arg substr instead of lvalue substr.
108. Make appropriate use of lvalue variables.
109. Use glob, not <>.
110. Avoid a raw select for non-integer sleeps.
111. Always use a block with a map and grep.
112. Use the ‘non-builtin’ builtins.

Subroutines
113. Call subroutines with parentheses but without a leading &.
114. Don’t give subroutines the same names as built-in functions.
115. Always unpack &_ first.
116. Use a hash of named arguments for any subroutine that has more than three parameters.
117. Use definedness or existence to test for missing arguments.
118. Resolve any default argument values as soon as $_ is unpacked.
119. Always return scalar in scalar returns.
120. Make list-returning subroutines return the ‘obvious’ value in scalar context.
121. When there is no ‘obvious’ scalar context return value, consider Contextual::Return instead.
122. Don’t use subroutine prototypes.
123. Always return via an explicit return.
124. Use a bare return to return failure.

I/O
125. Don’t use bareword filehandles.
126. Use indirect filehandles.
127. If you have to use a package filehandle, localize it first.
128. Use either the IO::File module or the three-argument form of open.
129. Never open, close, or print to a file without checking the outcome.
130. Close filehandles explicitly, and as soon as possible.
131. Use while (co) not for (co).
132. Prefer line-based I/O to slurping.
133. Slurp a filehandle with a do block for purity.
134. Slurp a stream with Perl6::Slurp for power and simplicity.
135. Avoid using *STDIN, unless you really mean it.
136. Always put filehandles in braces within any print statement.
137. Always prompt for interactive input.
138. Don't reinvent the standard test for interactivity.
139. Use the IO::Prompt module for prompting.
140. Always convey the progress of long non-interactive operations within interactive applications.
141. Consider using the Smart::Comments module to automate your progress indicators.
142. Avoid a raw select when setting autoflushes.

References

143. Wherever possible, dereference with arrows.
144. Where prefix dereferencing is unavoidable, put braces around the reference.
145. Never use symbolic references.
146. Use weaken to prevent circular data structures from leaking memory.

Regular Expressions

147. Always use the /x flag.
148. Always use the /m flag.
149. Use \A and \Z as string boundary anchors.
150. Use \A or \Z to indicate 'end of string'.
151. Always use the /s flag.
152. Consider mandating the Regexp::Autoflags module.
153. Use m(…) in preference to /…/ in multiline regexes.
154. Don't use any delimiters other than /…/ or m(…).
155. Prefer singular character classes to escaped metacharacters.
156. Prefer named characters to escaped metacharacters.
157. Prefer properties to enumerated character classes.
158. Consider matching arbitrary whitespace, rather than specific whitespace characters.
159. Be specific when matching 'as much as possible'.
160. Use capturing parentheses only when you intend to capture.
161. Use the numeric capture variables only when you're sure that the preceding match succeeded.
162. Always give captured substrings proper names.
163. Tokenize input using the /gc flag.
164. Build regular expressions from tables.
165. Build complex regular expressions from simpler pieces.
166. Consider using Regexp::Common instead of writing your own regexes.
167. Always use character classes instead of single-character alternations.
168. Factor out common affixes from alternations.
169. Prevent useless backtracking.
170. Prefer fixed-string eq comparisons to fixed-pattern regex matches.

Error Handling

171. Throw exceptions instead of returning special values or setting flags.
172. Make failure boils up exceptions too.
173. Make failures fatal in all contexts.
174. Be careful when testing for failure of the system builtin.
175. Throw exceptions on all failures, including recoverable ones.
176. Have exceptions report from the caller's location, not from the place where they were thrown.
177. Compose error messages in the recipient's dialect.
178. Document every error message in the recipient's dialect.
179. Use exception objects whenever failure data needs to be conveyed to a handler.
180. Use exception objects when two or more exceptions are related.
181. Catch exception objects in most-derived-first order.
182. Build exception classes automatically.
183. Unpack the exception variable in extended exception handlers.

Command-Line Processing

185. Enforce a single consistent command-line structure.
186. Adhere to a standard set of conventions in your command-line syntax.
187. Standardize your meta-options.
188. Allow the same filename to be specified for both input and output.
189. Standardize on a single approach to command-line processing.
190. Ensure that your interface, run-time messages, and documentation remain consistent.
191. Factor out command-line interface components into a shared module.

Objects

192. Make object orientation a choice, not a default.
193. Choose object orientation using appropriate criteria.
194. Don't use pseudohashes.
195. Don't use restricted hashes.
196. Always use fully encapsulated objects.
197. Give every constructor the same standard name.
198. Don't let a constructor clone objects.
199. Always provide a destructor for every inside-out class.
200. When creating methods, follow the general guidelines for subroutines.
201. Provide separate read and write accessors.
202. Don't use value accessors.
203. Don't use the indirect object syntax.
204. Provide an optimal interface, rather than a minimal one.
205. Overload only the isomorphic operators of algebraic classes.
206. Always consider overloading the boolean, numeric, and string coercions of objects.

Class Hierarchies

207. Don't manipulate the list of base classes directly.
208. Use distributed encapsulated objects.
209. Never use the one-argument form of bless.
210. Pass constructor arguments as named values, using a hash reference.
211. Distinguish arguments for base classes by class name as well.
212. Separate your construction, initialization, and destruction processes.
213. Build the standard class infrastructure automatically.
214. Use Class::Std to automate the deallloction of attribute data.