

```

1 2015/09/21 - HAProxy coding style - Willy Tarreau <@wt.eu>
2 -----
3
4 A number of contributors are often embarrassed with coding style issues, they
5 don't always know if they're doing it right, especially since the coding style
6 has elvolved along the years. What is explained here is not necessarily what is
7 applied in the code, but new code should as much as possible conform to this
8 style. Coding style fixes happen when code is replaced. It is useless to send
9 patches to fix coding style only, they will be rejected, unless they belong to
10 a patch series which needs these fixes prior to get code changes. Also, please
11 avoid fixing coding style in the same patches as functional changes, they make
12 code review harder.
13
14 A good way to quickly validate your patch before submitting it is to pass it
15 through the Linux kernel's checkpatch.pl utility which can be downloaded here :
16
17 http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/scripts/check...
18
19 Running it with the following options relaxes its checks to accommodate to the
20 extra degree of freedom that is tolerated in HAProxy's coding style compared to
21 the stricter style used in the kernel :
22
23 checkpatch.pl -q --max-line-length=160 --no-tree --no-signoff \
24 --ignore=LEADING_SPACE,CODE_INDENT,DEEP_INDENTATION \
25 --ignore=ELSE_AFTER_BRACE < patch
26
27 You can take its output as hints instead of strict rules, but in general its
28 output will be accurate and it may even spot some real bugs.
29
30 When modifying a file, you must accept the terms of the license of this file
31 which is recalled at the top of the file, or is explained in the LICENSE file,
32 or if not stated, defaults to LGPL version 2.1 or later for files in the
33 'include' directory, and GPL version 2 or later for all other files.
34
35 When adding a new file, you must add a copyright banner at the top of the
36 file with your real name, e-mail address and a reminder of the license.
37 Contributions under incompatible licenses or too restrictive licenses might
38 get rejected. If in doubt, please apply the principle above for existing files.
39
40 All code examples below will intentionally be prefixed with " | " to mark
41 where the code aligns with the first column, and tabs in this document will be
42 represented as a series of 8 spaces so that it displays the same everywhere.
43
44
45 1) Indentation and alignment
46 -----
47
48 1.1) Indentation
49 -----
50
51 Indentation and alignment are two completely different things that people often
52 get wrong. Indentation is used to mark a sub-level in the code. A sub-level
53 means that a block is executed in the context of another block (eg: a function
54 or a condition) :
55
56 | main(int argc, char **argv)
57 | {
58 |     int i;
59 |
60 |     if (argc < 2)
61 |         exit(1);
62 | }
63
64 In the example above, the code belongs to the main() function and the exit()
65 call belongs to the if statement. Indentation is made with tabs (\t, ASCII 9),

```

```

66 which allows any developer to configure their preferred editor to use their
67 own tab size and to still get the text properly indented. Exactly one tab is
68 used per sub-level. Tabs may only appear at the beginning of a line or after
69 another tab. It is illegal to put a tab after some text, as it mangles displays
70 in a different manner for different users (particularly when used to align
71 comments or values after a #define). If you're tempted to put a tab after some
72 text, then you're doing it wrong and you need alignment instead (see below).
73
74 Note that there are places where the code was not properly indented in the
75 past. In order to view it correctly, you may have to set your tab size to 8
76 characters.
77
78
79 1.2) Alignment
80 -----
81
82 Alignment is used to continue a line in a way to makes things easier to group
83 together. By definition, alignment is character-based, so it uses spaces. Tabs
84 would not work because for one tab there would not be as many characters on all
85 displays. For instance, the arguments in a function declaration may be broken
86 into multiple lines using alignment spaces :
87
88 | int http_header_match2(const char *hdr, const char *end,
89 |                         const char *name, int len)
90 | {
91 |     ...
92 | }
93
94 In this example, the "const char *name" part is aligned with the first
95 character of the group it belongs to (list of function arguments). Placing it
96 here makes it obvious that it's one of the function's arguments. Multiple lines
97 are easy to handle this way. This is very common with long conditions too :
98
99 | if ((len < eol - sol) &&
100 |     (sol[len] == ':') &&
101 |     (strncasecmp(sol, name, len) == 0)) {
102 |     ctx->del = len;
103 | }
104
105 If we take again the example above marking tabs with "[-Tabs-]" and spaces
106 with "#", we get this :
107
108 | [-Tabs-]if ((len < eol - sol) &&
109 | [-Tabs-]####(sol[len] == ':') &&
110 | [-Tabs-]####(strncasecmp(sol, name, len) == 0)) {
111 | [-Tabs-][-Tabs-]ctx->del = len;
112 | [-Tabs-]}
113
114 It is worth noting that some editors tend to confuse indentations and alignment.
115 Emacs is notoriously known for this brokenness, and is responsible for almost
116 all of the alignment mess. The reason is that Emacs only counts spaces, tries
117 to fill as many as possible with tabs and completes with spaces. Once you know
118 it, you just have to be careful, as alignment is not used much, so generally it
119 is just a matter of replacing the last tab with 8 spaces when this happens.
120
121 Indentation should be used everywhere there is a block or an opening brace. It
122 is not possible to have two consecutive closing braces on the same column, it
123 means that the innermost was not indented.
124
125 Right :
126
127 | main(int argc, char **argv)
128 | {
129 |     if (argc > 1) {
130 |         printf("Hello\n");

```

```

131 |         }
132 |         exit(0);
133 |     }
134 |
135 | Wrong :
136 |
137 | main(int argc, char **argv)
138 | {
139 |     if (argc > 1) {
140 |         printf("Hello\n");
141 |     }
142 |     exit(0);
143 | }
144 |

```

A special case applies to switch/case statements. Due to my editor's settings, I've been used to align "case" with "switch" and to find it somewhat logical since each of the "case" statements opens a sublevel belonging to the "switch" statement. But indenting "case" after "switch" is accepted too. However in any case, whatever follows the "case" statement must be indented, whether or not it contains braces :

```

151 | switch (*arg) {
152 | case 'A': {
153 |     int i;
154 |     for (i = 0; i < 10; i++)
155 |         printf("Please stop pressing 'A'\n");
156 |     break;
157 | }
158 | case 'B':
159 |     printf("You pressed 'B'\n");
160 |     break;
161 | case 'C':
162 | case 'D':
163 |     printf("You pressed 'C' or 'D'\n");
164 |     break;
165 | default:
166 |     printf("I don't know what you pressed\n");
167 | }
168 |
169 |
170 |
171 | 2) Braces
172 | -----
173 |
174 | Braces are used to delimit multiple-instruction blocks. In general it is
175 | preferred to avoid braces around single-instruction blocks as it reduces the
176 | number of lines :
177 |
178 | Right :
179 |
180 | if (argc >= 2)
181 |     exit(0);
182 |
183 |
184 | Wrong :
185 |
186 | if (argc >= 2) {
187 |     exit(0);
188 | }
189 |
190 | But it is not that strict, it really depends on the context. It happens from
191 | time to time that single-instruction blocks are enclosed within braces because
192 | it makes the code more symmetrical, or more readable. Example :
193 |
194 | if (argc < 2) {
195 |     printf("Missing argument\n");
196 |     exit(1);
197 | }

```

```

196 |     } else {
197 |         exit(0);
198 |     }
199 |
200 | Braces are always needed to declare a function. A function's opening brace must
201 | be placed at the beginning of the next line :
202 |
203 | Right :
204 |
205 | int main(int argc, char **argv)
206 | {
207 |     exit(0);
208 | }
209 |
210 | Wrong :
211 |
212 | int main(int argc, char **argv) {
213 |     exit(0);
214 | }
215 |

```

Note that a large portion of the code still does not conforms to this rule, as it took years to get all authors to adapt to this more common standard which is now preferred, as it avoids visual confusion when function declarations are broken on multiple lines :

```

221 | Right :
222 |
223 | int foo(const char *hdr, const char *end,
224 |         const char *name, const char *err,
225 |         int len)
226 | {
227 |     int i;
228 |
229 | Wrong :
230 |
231 | int foo(const char *hdr, const char *end,
232 |         const char *name, const char *err,
233 |         int len) {
234 |     int i;
235 |

```

Braces should always be used where there might be an ambiguity with the code later. The most common example is the stacked "if" statement where an "else" may be added later at the wrong place breaking the code, but it also happens with comments or long arguments in function calls. In general, if a block is more than one line long, it should use braces.

Dangerous code waiting of a victim :

```

243 | if (argc < 2)
244 |     /* ret must not be negative here */
245 |     if (ret < 0)
246 |         return -1;
247 |
248 |
249 | Wrong change :
250 |
251 | if (argc < 2)
252 |     /* ret must not be negative here */
253 |     if (ret < 0)
254 |         return -1;
255 |     else
256 |         return 0;
257 |

```

It will do this instead of what your eye seems to tell you :

```

258 | if (argc < 2)
259 |     if (argc < 2)
260 |         if (argc < 2)

```

```

261 | | /* ret must not be negative here */
262 | | if (ret < 0)
263 | |     return -1;
264 | | else
265 | |     return 0;
266 |
267 | Right :
268 |
269 | | if (argc < 2) {
270 | |     /* ret must not be negative here */
271 | |     if (ret < 0)
272 | |         return -1;
273 | | }
274 | | else
275 | |     return 0;
276 |
277 | Similarly dangerous example :
278 |
279 | | if (ret < 0)
280 | |     /* ret must not be negative here */
281 | |     complain();
282 | | init();
283 |
284 | Wrong change to silent the annoying message :
285 |
286 | | if (ret < 0)
287 | |     /* ret must not be negative here */
288 | |     //complain();
289 | | init();
290 |
291 | ... which in fact means :
292 |
293 | | if (ret < 0)
294 | |     init();
295 |

```

3) Breaking lines

```

-----

```

There is no strict rule for line breaking. Some files try to stick to the 80 column limit, but given that various people use various tab sizes, it does not make much sense. Also, code is sometimes easier to read with less lines, as it represents less surface on the screen (since each new line adds its tabs and spaces). The rule is to stick to the average line length of other lines. If you are working in a file which fits in 80 columns, try to keep this goal in mind. If you're in a function with 120-chars lines, there is no reason to add many short lines, so you can make longer lines.

In general, opening a new block should lead to a new line. Similarly, multiple instructions should be avoided on the same line. But some constructs make it more readable when those are perfectly aligned :

A copy-paste bug in the following construct will be easier to spot :

```

313 | if (omult % idiv == 0) { omult /= idiv; idiv = 1; }
314 |
315 | if (idiv % omult == 0) { idiv /= omult; omult = 1; }
316 |
317 | if (imult % odiv == 0) { imult /= odiv; odiv = 1; }
318 |
319 | if (odiv % imult == 0) { odiv /= imult; imult = 1; }

```

than in this one :

```

320 |
321 |
322 | if (omult % idiv == 0) {
323 |     omult /= idiv;
324 |     idiv = 1;
325 | }

```

```

326 | | if (idiv % omult == 0) {
327 | |     idiv /= omult;
328 | |     omult = 1;
329 | | }
330 | | if (imult % odiv == 0) {
331 | |     imult /= odiv;
332 | |     odiv = 1;
333 | | }
334 | | if (odiv % imult == 0) {
335 | |     odiv /= imult;
336 | |     imult = 1;
337 | | }
338 |

```

What is important is not to mix styles. For instance there is nothing wrong with having many one-line "case" statements as long as most of them are this short like below :

```

342 | | switch (*arg) {
343 | |     case 'A': ret = 1; break;
344 | |     case 'B': ret = 2; break;
345 | |     case 'C': ret = 4; break;
346 | |     case 'D': ret = 8; break;
347 | |     default : ret = 0; break;
348 | | }
349 |
350 |

```

Otherwise, prefer to have the "case" statement on its own line as in the example in section 1.2 about alignment. In any case, avoid to stack multiple control statements on the same line, so that it will never be the needed to add two tab levels at once :

Right :

```

357 | | switch (*arg) {
358 | |     case 'A':
359 | |         if (ret < 0)
360 | |             ret = 1;
361 | |         break;
362 | |     default : ret = 0; break;
363 | | }
364 |

```

Wrong :

```

365 | | switch (*arg) {
366 | |     case 'A': if (ret < 0)
367 | |         ret = 1;
368 | |         break;
369 | |     default : ret = 0; break;
370 | | }
371 |
372 |
373 |

```

Right :

```

374 | | if (argc < 2)
375 | |     if (ret < 0)
376 | |         return -1;
377 |
378 |
379 |

```

or Right :

```

381 | | if (argc < 2)
382 | |     if (ret < 0) return -1;
383 |
384 |
385 |

```

but Wrong :

```

386 | | if (argc < 2) if (ret < 0) return -1;
387 |
388 |
389 |
390 |

```

391 When complex conditions or expressions are broken into multiple lines, please
 392 do ensure that alignment is perfectly appropriate, and group all main operators
 393 on the same side (which you're free to choose as long as it does not change for
 394 every block. Putting binary operators on the right side is preferred as it does
 395 not mangle with alignment but various people have their preferences.
 396

```

397 Right :
398
399 | if ((txn->flags & TX_NOT_FIRST) &&
400 | ((req->flags & BF_FULL) ||
401 | req->r < req->lr ||
402 | req->r > req->data + req->size - global.tune.maxrewrite)) {
403 |     return 0;
404 | }
405
406 Right :
407
408 | if ((txn->flags & TX_NOT_FIRST)
409 | && ((req->flags & BF_FULL)
410 | || req->r < req->lr
411 | || req->r > req->data + req->size - global.tune.maxrewrite)) {
412 |     return 0;
413 | }
414
415 Wrong :
416
417 | if ((txn->flags & TX_NOT_FIRST) &&
418 | ((req->flags & BF_FULL) ||
419 | req->r < req->lr
420 | || req->r > req->data + req->size - global.tune.maxrewrite)) {
421 |     return 0;
422 | }

```

423
 424 If it makes the result more readable, parenthesis may even be closed on their
 425 own line in order to align with the opening one. Note that should normally not
 426 be needed because such code would be too complex to be digged into.
 427

428 The "else" statement may either be merged with the closing "if" brace or lie on
 429 its own line. The later is preferred but it adds one extra line to each control
 430 block which is annoying in short ones. However, if the "else" is followed by an
 431 "if", then it should really be on its own line and the rest of the if/else
 432 blocks must follow the same style.
 433

```

434 Right :
435
436 | if (a < b) {
437 |     return a;
438 | }
439 | else {
440 |     return b;
441 | }
442
443 Right :
444
445 | if (a < b) {
446 |     return a;
447 | } else {
448 |     return b;
449 | }
450
451 Right :
452
453 | if (a < b) {
454 |     return a;
455 | }

```

```

456 | else if (a != b) {
457 |     return b;
458 | }
459 | else {
460 |     return 0;
461 | }
462
463 Wrong :
464
465 | if (a < b) {
466 |     return a;
467 | } else if (a != b) {
468 |     return b;
469 | } else {
470 |     return 0;
471 | }
472
473 Wrong :
474
475 | if (a < b) {
476 |     return a;
477 | }
478 | else if (a != b) {
479 |     return b;
480 | } else {
481 |     return 0;
482 | }
483
484 4) Spacing
485 -----
486
487
488
489
490
491
492

```

493 Correctly spacing code is very important. When you have to spot a bug at 3am,
 494 you need it to be clear. When you expect other people to review your code, you
 495 want it to be clear and don't want them to get nervous when trying to find what
 496 you did.
 497

498 Always place spaces around all binary or ternary operators, commas, as well as
 499 after semi-colons and opening braces if the line continues :

```

499 Right :
500
501 | int ret = 0;
502 | /* if (x >> 4) { x >= 4; ret += 4; } */
503 | ret += (x >> 4) ? (x >= 4, 4) : 0;
504 | val = ret + ((0xFFFFFAA50U >> (x << 1)) & 3) + 1;
505
506 Wrong :
507
508 | int ret=0;
509 | /* if (x>>4) {x>=4;ret+=4;} */
510 | ret+=(x>>4)?(x>=4,4):0;
511 | val=ret+((0xFFFFFAA50U>>(x<<1))&3)+1;
512
513 Never place spaces after unary operators (&, *, -, !, ~, ++, --) nor cast, as
514 they might be confused with they binary counterpart, nor before commas or
515 semicolons :
516
517 Right :
518
519 | bit = !!(~len++ ^ -(unsigned char)*x);
520
521 Wrong :
522
523 | bit = ! ! (~len++ ^ - (unsigned char) * x) ;

```

```

521 Note that "sizeof" is a unary operator which is sometimes considered as a
522 language keyword, but in no case it is a function. It does not require
523 parenthesis so it is sometimes followed by spaces and sometimes not when
524 there are no parenthesis. Most people do not really care as long as what
525 is written is unambiguous.
526
527 Braces opening a block must be preceeded by one space unless the brace is
528 placed on the first column :
529
530 Right :
531 | if (argc < 2) {
532 | }
533
534 Wrong :
535 | if (argc < 2){
536 | }
537
538 Do not add unneeded spaces inside parenthesis, they just make the code less
539 readable.
540
541 Right :
542 | if (x < 4 && (!y || !z))
543 |     break;
544
545 Wrong :
546 | if ( x < 4 && ( !y || !z ) )
547 |     break;
548
549 Language keywords must all be followed by a space. This is true for control
550 statements (do, for, while, if, else, return, switch, case), and for types
551 (int, char, unsigned). As an exception, the last type in a cast does not take
552 a space before the closing parenthesis). The "default" statement in a "switch"
553 construct is generally just followed by the colon. However the colon after a
554 "case" or "default" statement must be followed by a space.
555
556 Right :
557 | if (nbargs < 2) {
558 |     printf("Missing arg at %c\n", *(char *)ptr);
559 |     for (i = 0; i < 10; i++) beep();
560 |     return 0;
561 | }
562 | switch (*arg) {
563 |
564 |     printf("Missing arg at %c\n", *(char *)ptr);
565 |     for (i = 0; i < 10; i++) beep();
566 |     return 0;
567 | }
568 | switch (*arg) {
569 |
570 |     if (nbargs < 2){
571 |         printf("Missing arg at %c\n", *(char *)ptr);
572 |         for (i = 0; i < 10; i++)beep();
573 |         return 0;
574 |     }
575 |     switch(*arg) {
576 |
577 |
578 |
579 Function calls are different, the opening parenthesis is always coupled to the
580 function name without any space. But spaces are still needed after commas :
581
582 Right :
583 | if (!init(argc, argv))
584 |     exit(1);
585

```

```

586 Wrong :
587
588 | if (!init (argc,argv))
589 |     exit(1);
590
591
592 5) Excess or lack of parenthesis
593 -----
594
595 Sometimes there are too many parenthesis in some formulas, sometimes there are
596 too few. There are a few rules of thumb for this. The first one is to respect
597 the compiler's advice. If it emits a warning and asks for more parenthesis to
598 avoid confusion, follow the advice at least to shut the warning. For instance,
599 the code below is quite ambiguous due to its alignment :
600
601 | if (var1 < 2 || var2 < 2 &&
602 |     var3 != var4) {
603 |     /* fail */
604 |     return -3;
605 | }
606
607 Note that this code does :
608
609 | if (var1 < 2 || (var2 < 2 && var3 != var4)) {
610 |     /* fail */
611 |     return -3;
612 | }
613
614 But maybe the author meant :
615
616 | if ((var1 < 2 || var2 < 2) && var3 != var4) {
617 |     /* fail */
618 |     return -3;
619 | }
620
621 A second rule to put parenthesis is that people don't always know operators
622 precedence too well. Most often they have no issue with operators of the same
623 category (eg: booleans, integers, bit manipulation, assignment) but once these
624 operators are mixed, it causes them all sort of issues. In this case, it is
625 wise to use parenthesis to avoid errors. One common error concerns the bit
626 shift operators because they're used to replace multiplies and divides but
627 don't have the same precedence :
628
629 The expression :
630
631 | x = y * 16 + 5;
632
633 becomes :
634
635 | x = y << 4 + 5;
636
637 which is wrong because it is equivalent to :
638
639 | x = y << (4 + 5);
640
641 while the following was desired instead :
642
643 | x = (y << 4) + 5;
644
645 It is generally fine to write boolean expressions based on comparisons without
646 any parenthesis. But on top of that, integer expressions and assignments should
647 then be protected. For instance, there is an error in the expression below
648 which should be safely rewritten :
649
650

```

```

651 Wrong :
652
653 | if (var1 > 2 && var1 < 10 ||
654 |   var1 > 2 + 256 && var2 < 10 + 256 ||
655 |   var1 > 2 + 1 << 16 && var2 < 10 + 2 << 16)
656 |   return 1;
657
658 Right (may remove a few parenthesis depending on taste) :
659
660 | if ((var1 > 2 && var1 < 10) ||
661 |   (var1 > (2 + 256) && var2 < (10 + 256)) ||
662 |   (var1 > (2 + (1 << 16)) && var2 < (10 + (1 << 16))))
663 |   return 1;
664

```

The "return" statement is not a function, so it takes no argument. It is a control statement which is followed by the expression to be returned. It does not need to be followed by parenthesis :

```

669 Wrong :
670
671 | int ret0()
672 | {
673 |   return(0);
674 | }
675
676 Right :
677
678 | int ret0()
679 | {
680 |   return 0;
681 | }
682

```

Parenthesis are also found in type casts. Type casting should be avoided as much as possible, especially when it concerns pointer types. Casting a pointer disables the compiler's type checking and is the best way to get caught doing wrong things with data not the size you expect. If you need to manipulate multiple data types, you can use a union instead. If the union is really not convenient and casts are easier, then try to isolate them as much as possible, for instance when initializing function arguments or in another function. Not proceeding this way causes huge risks of not using the proper pointer without any notification, which is especially true during copy-pastes.

```

693 Wrong :
694
695 | void *check_private_data(void *arg1, void *arg2)
696 | {
697 |   char *area;
698
699 |   if (*(int *)arg1 > 1000)
700 |     return NULL;
701 |   if (memcmp(*(const char *)arg2, "send(", 5) != 0))
702 |     return NULL;
703 |   area = malloc(*(int *)arg1);
704 |   if (!area)
705 |     return NULL;
706 |   memcpy(area, *(const char *)arg2 + 5, *(int *)arg1);
707 |   return area;
708 | }
709
710 Right :
711
712 | void *check_private_data(void *arg1, void *arg2)
713 | {
714 |   char *area;
715 |   int len = *(int *)arg1;

```

```

716 |   const char *msg = arg2;
717 |
718 |   if (len > 1000)
719 |     return NULL;
720 |   if (memcmp(msg, "send(", 5) != 0)
721 |     return NULL;
722 |   area = malloc(len);
723 |   if (!area)
724 |     return NULL;
725 |   memcpy(area, msg + 5, len);
726 |   return area;
727 | }
728

```

6) Ambiguous comparisons with zero or NULL

In C, '0' has no type, or it has the type of the variable it is assigned to. Comparing a variable or a return value with zero means comparing with the representation of zero for this variable's type. For a boolean, zero is false. For a pointer, zero is NULL. Very often, to make things shorter, it is fine to use the '!' unary operator to compare with zero, as it is shorter and easier to remind or understand than a plain '0'. Since the '!' operator is read "not", it helps read code faster when what follows it makes sense as a boolean, and it is often much more appropriate than a comparison with zero which makes an equal sign appear at an undesirable place. For instance :

```

741 | if (!isdigit(*c) && !isspace(*c))
742 |   break;
743

```

is easier to understand than :

```

747 | if (isdigit(*c) == 0 && isspace(*c) == 0)
748 |   break;
749

```

For a char this "not" operator can be reminded as "no remaining char", and the absence of comparison to zero implies existence of the tested entity, hence the simple strcpy() implementation below which automatically stops once the last zero is copied :

```

755 | void my_strcpy(char *d, const char *s)
756 | {
757 |   while ((*d++ = *s++));
758 | }
759

```

Note the double parenthesis in order to avoid the compiler telling us it looks like an equality test.

For a string or more generally any pointer, this test may be understood as an existence test or a validity test, as the only pointer which will fail to validate equality is the NULL pointer :

```

767 | area = malloc(1000);
768 | if (!area)
769 |   return -1;
770

```

However sometimes it can fool the reader. For instance, strcmp() precisely is one of such functions whose return value can make one think the opposite due to its name which may be understood as "if strings compare...". Thus it is strongly recommended to perform an explicit comparison with zero in such a case, and it makes sense considering that the comparison's operator is the same that is wanted to compare the strings (note that current config parser lacks a lot in this regards) :

```

780 | strcmp(a, b) == 0 <=> a == b

```

```

781 strcmp(a, b) != 0 <=> a != b
782 strcmp(a, b) < 0 <=> a < b
783 strcmp(a, b) > 0 <=> a > b
784
785 Avoid this :
786
787 | if (strcmp(arg, "test"))
788 |     printf("this is not a test\n");
789
790 | if (!strcmp(arg, "test"))
791 |     printf("this is a test\n");
792
793 Prefer this :
794
795 | if (strcmp(arg, "test") != 0)
796 |     printf("this is not a test\n");
797
798 | if (strcmp(arg, "test") == 0)
799 |     printf("this is a test\n");
800
801
802
803
804

```

7) System call returns

This is not directly a matter of coding style but more of bad habits. It is important to check for the correct value upon return of syscalls. The proper return code indicating an error is described in its man page. There is no reason to consider wider ranges than what is indicated. For instance, it is common to see such a thing :

```

809 | if ((fd = open(file, O_RDONLY)) < 0)
810 |     return -1;
811
812
813

```

This is wrong. The man page says that -1 is returned if an error occurred. It does not suggest that any other negative value will be an error. It is possible that a few such issues have been left in existing code. They are bugs for which fixes are accepted, even though they're currently harmless since open() is not known for returning negative values at the moment.

8) Declaring new types, names and values

Please refrain from using "typedef" to declare new types, they only obfuscate the code. The reader never knows whether he's manipulating a scalar type or a struct. For instance it is not obvious why the following code fails to build :

```

827 | int delay_expired(timer_t exp, timer_us_t now)
828 | {
829 |     return now >= exp;
830 | }
831
832
833

```

With the types declared in another file this way :

```

834 | typedef unsigned int timer_t;
835 | typedef struct timeval timer_us_t;
836
837
838

```

This cannot work because we're comparing a scalar with a struct, which does not make sense. Without a typedef, the function would have been written this way without any ambiguity and would not have failed :

```

841 | int delay_expired(unsigned int exp, struct timeval *now)
842 | {
843 |     return now >= exp->tv_sec;
844 | }
845

```

Declaring special values may be done using enums. Enums are a way to define structured integer values which are related to each other. They are perfectly suited for state machines. While the first element is always assigned the zero value, not everybody knows that, especially people working with multiple languages all the day. For this reason it is recommended to explicitly force the first value even if it's zero. The last element should be followed by a comma if it is planned that new elements might later be added, this will make later patches shorter. Conversely, if the last element is placed in order to get the number of possible values, it must not be followed by a comma and must be preceded by a comment :

```

846
847 | enum {
848 |     first = 0,
849 |     second,
850 |     third,
851 |     fourth,
852 | };
853
854
855 | enum {
856 |     first = 0,
857 |     second,
858 |     third,
859 |     fourth,
860 |     /* nbvalues must always be placed last */
861 |     nbvalues
862 | };
863
864
865
866
867
868
869
870
871
872
873
874

```

Structure names should be short enough not to mangle function declarations, and explicit enough to avoid confusion (which is the most important thing).

Wrong :

```

875 | struct request_args { /* arguments on the query string */
876 |     char *name;
877 |     char *value;
878 |     struct misc_args *next;
879 | };
880
881
882
883
884
885
886
887

```

Right :

```

888 | struct qs_args { /* arguments on the query string */
889 |     char *name;
890 |     char *value;
891 |     struct qs_args *next;
892 | };
893
894
895
896
897
898
899
900

```

When declaring new functions or structures, please do not use CamelCase, which is a style where upper and lower case are mixed in a single word. It causes a lot of confusion when words are composed from acronyms, because it's hard to stick to a rule. For instance, a function designed to generate an ISN (initial sequence number) for a TCP/IP connection could be called :

```

901 | - generateTcpiPisn()
902 | - generateTcpiPisn()
903 | - generateTcpiPisn()
904 | - generateTcpiPisn()
905 | etc...
906

```

None is right, none is wrong, these are just preferences which might change along the code. Instead, please use an underscore to separate words. Lowercase is preferred for the words, but if acronyms are upcased it's not dramatic. The real advantage of this method is that it creates unambiguous levels even for

```

911 short names.
912
913 Valid examples :
914
915 - generate_tcpip_isn()
916 - generate_tcp_ip_isn()
917 - generate_TCPIP_ISN()
918 - generate_TCP_IP_ISN()
919
920 Another example is easy to understand when 3 arguments are involved in naming
921 the function :
922
923 Wrong (naming conflict) :
924
925 | /* returns A + B * C */
926 | int mulABC(int a, int b, int c)
927 | {
928 |     return a + b * c;
929 | }
930
931 | /* returns (A + B) * C */
932 | int mulABC(int a, int b, int c)
933 | {
934 |     return (a + b) * c;
935 | }
936
937 Right (unambiguous naming) :
938
939 | /* returns A + B * C */
940 | int mul_a_bc(int a, int b, int c)
941 | {
942 |     return a + b * c;
943 | }
944
945 | /* returns (A + B) * C */
946 | int mul_ab_c(int a, int b, int c)
947 | {
948 |     return (a + b) * c;
949 | }
950
951 Whenever you manipulate pointers, try to declare them as "const", as it will
952 save you from many accidental misuses and will only cause warnings to be
953 emitted when there is a real risk. In the examples below, it is possible to
954 call my_strcpy() with a const string only in the first declaration. Note that
955 people who ignore "const" are often the ones who cast a lot and who complain
956 from segfaults when using strtok() !
957
958 Right :
959
960 | void my_strcpy(char *d, const char *s)
961 | {
962 |     while ((*d++ = *s++));
963 | }
964
965 | void say_hello(char *dest)
966 | {
967 |     my_strcpy(dest, "hello\n");
968 | }
969
970 Wrong :
971
972 | void my_strcpy(char *d, char *s)
973 | {
974 |     while ((*d++ = *s++));
975 | }

```

```

976 |
977 | void say_hello(char *dest)
978 | {
979 |     my_strcpy(dest, "hello\n");
980 | }
981
982
983 9) Getting macros right
984 -----
985
986 It is very common for macros to do the wrong thing when used in a way their
987 author did not have in mind. For this reason, macros must always be named with
988 uppercase letters only. This is the only way to catch the developer's eye when
989 using them, so that he double-checks whether he's taking risks or not. First,
990 macros must never ever be terminated by a semi-colon, or they will close the
991 wrong block once in a while. For instance, the following will cause a build
992 error before the "else" due to the double semi-colon :
993
994 Wrong :
995
996 | #define WARN printf("warning\n");
997 | ...
998 |     if (a < 0)
999 |         WARN;
1000 |     else
1001 |         a--;
1002
1003 Right :
1004
1005 | #define WARN printf("warning\n")
1006
1007 If multiple instructions are needed, then use a do { } while (0) block, which
1008 is the only construct which respects *exactly* the semantics of a single
1009 instruction :
1010
1011 | #define WARN do { printf("warning\n"); log("warning\n"); } while (0)
1012 | ...
1013 |
1014 |     if (a < 0)
1015 |         WARN;
1016 |     else
1017 |         a--;
1018
1019 Second, do not put unprotected control statements in macros, they will
1020 definitely cause bugs :
1021
1022 Wrong :
1023
1024 | #define WARN if (verbose) printf("warning\n")
1025 | ...
1026 |     if (a < 0)
1027 |         WARN;
1028 |     else
1029 |         a--;
1030
1031 Which is equivalent to the undesired form below :
1032
1033 |     if (a < 0)
1034 |         if (verbose)
1035 |             printf("warning\n");
1036 |     else
1037 |         a--;
1038
1039 Right way to do it :
1040

```



```
1041 | #define WARN do { if (verbose) printf("warning\n"); } while (0)
1042 | ...
1043 |     if (a < 0)
1044 |         WARN;
1045 |     else
1046 |         a--;
1047 |
1048 | Which is equivalent to :
1049 |
1050 |     if (a < 0)
1051 |         do { if (verbose) printf("warning\n"); } while (0);
1052 |     else
1053 |         a--;
1054 |
1055 | Macro parameters must always be surrounded by parenthesis, and must never be
1056 | duplicated in the same macro unless explicitly stated. Also, macros must not be
1057 | defined with operators without surrounding parenthesis. The MIN/MAX macros are
1058 | a pretty common example of multiple misuses, but this happens as early as when
1059 | using bit masks. Most often, in case of any doubt, try to use inline functions
1060 | instead.
1061 |
1062 | Wrong :
1063 |
1064 | #define MIN(a, b) a < b ? a : b
1065 |
1066 | /* returns 2 * min(a,b) + 1 */
1067 | int double_min_pl(int a, int b)
1068 | {
1069 |     return 2 * MIN(a, b) + 1;
1070 | }
1071 |
1072 | What this will do :
1073 |
1074 | int double_min_pl(int a, int b)
1075 | {
1076 |     return 2 * a < b ? a : b + 1;
1077 | }
1078 |
1079 | Which is equivalent to :
1080 |
1081 | int double_min_pl(int a, int b)
1082 | {
1083 |     return (2 * a) < b ? a : (b + 1);
1084 | }
1085 |
1086 | The first thing to fix is to surround the macro definition with parenthesis to
1087 | avoid this mistake :
1088 |
1089 | #define MIN(a, b) (a < b ? a : b)
1090 |
1091 | But this is still not enough, as can be seen in this example :
1092 |
1093 | /* compares either a or b with c */
1094 | int min_ab_c(int a, int b, int c)
1095 | {
1096 |     return MIN(a ? a : b, c);
1097 | }
1098 |
1099 | Which is equivalent to :
1100 |
1101 | int min_ab_c(int a, int b, int c)
1102 | {
1103 |     return (a ? a : b < c ? a : b : c);
1104 | }
1105 |
```

```
1106 | Which in turn means a totally different thing due to precedence :
1107 |
1108 | int min_ab_c(int a, int b, int c)
1109 | {
1110 |     return (a ? a : ((b < c) ? (a ? a : b) : c));
1111 | }
1112 |
1113 | This can be fixed by surrounding *each* argument in the macro with parenthesis:
1114 |
1115 | #define MIN(a, b) ((a) < (b) ? (a) : (b))
1116 |
1117 | But this is still not enough, as can be seen in this example :
1118 |
1119 | int min_apl_b(int a, int b)
1120 | {
1121 |     return MIN(++a, b);
1122 | }
1123 |
1124 | Which is equivalent to :
1125 |
1126 | int min_apl_b(int a, int b)
1127 | {
1128 |     return ((++a) < (b) ? (++a) : (b));
1129 | }
1130 |
1131 | Again, this is wrong because "a" is incremented twice if below b. The only way
1132 | to fix this is to use a compound statement and to assign each argument exactly
1133 | once to a local variable of the same type :
1134 |
1135 | #define MIN(a, b) ({ typeof(a) __a = (a); typeof(b) __b = (b); \
1136 |     (__a) < (__b) ? (__a) : (__b); \
1137 | })
1138 |
1139 | At this point, using static inline functions is much cleaner if a single type
1140 | is to be used :
1141 |
1142 | static inline int min(int a, int b)
1143 | {
1144 |     return a < b ? a : b;
1145 | }
1146 |
1147 |
1148 |
1149 | 10) Includes
1150 | -----
1151 | Includes are as much as possible listed in alphabetically ordered groups :
1152 | - the libc-standard includes (those without any path component)
1153 | - the includes more or less system-specific (sys/*, netinet/*, ...)
1154 | - includes from the local "common" subdirectory
1155 | - includes from the local "types" subdirectory
1156 | - includes from the local "proto" subdirectory
1157 |
1158 | Each section is just visually delimited from the other ones using an empty
1159 | line. The two first ones above may be merged into a single section depending on
1160 | developer's preference. Please do not copy-paste includes statements from other
1161 | files. Having too many includes significantly increases build time and makes it
1162 | hard to find which ones are needed later. Just include what you need and if
1163 | possible in alphabetical order so that when something is missing, it becomes
1164 | obvious where to look for it and where to add it.
1165 |
1166 | All files should include <common/config.h> because this is where build options
1167 | are prepared.
1168 |
1169 | Header files are split in two directories ("types" and "proto") depending on
1170 | what they provide. Types, structures, enums and #defines must go into the
```

1171 "types" directory. Function prototypes and inlined functions must go into the
 1172 "proto" directory. This split is because of inlined functions which
 1173 cross-reference types from other files, which cause a chicken-and-egg problem
 1174 if the functions and types are declared at the same place.

1175 All headers which do not depend on anything currently go to the "common"
 1176 subdirectory, but are equally well placed into the "proto" directory. It is
 1177 possible that one day the "common" directory will disappear.

1178 Include files must be protected against multiple inclusion using the common
 1179 #ifndef/#define/#endif trick with a tag derived from the include file and its
 1180 location.

1181 Comments

1182 -----

1183 Comments are preferably of the standard 'C' form using /* */. The C++ form `"/`
 1184 are tolerated for very short comments (eg: a word or two) but should be avoided
 1185 as much as possible. Multi-line comments are made with each intermediate line
 1186 starting with a star aligned with the first one, as in this example :

```
1187 | /*  

1188 |  * This is a multi-line  

1189 |  * comment.  

1190 |  */
```

1191 If multiple code lines need a short comment, try to align them so that you can
 1192 have multi-line sentences. This is rarely needed, only for really complex
 1193 constructs.

1201 Do not tell what you're doing in comments, but explain why you're doing it if
 1202 it seems not to be obvious. Also *do* indicate at the top of function what they
 1203 accept and what they don't accept. For instance, `strcpy()` only accepts output
 1204 buffers at least as large as the input buffer, and does not support any NULL
 1205 pointer. There is nothing wrong with that if the caller knows it.

1206 Wrong use of comments :

```
1207 | int flsnz8(unsigned int x)  

1208 | {  

1209 |     int ret = 0;  

1210 |     if (x >> 4) { x >=> 4; ret += 4; } /* initialize ret */  

1211 |     return ret + ((0xFFFFFAA50U >> (x << 1)) & 3) + 1; /* add ??? */  

1212 | }  

1213 | ...  

1214 | bit = ~len + (skip << 3) + 9; /* update bit */
```

1215 Right use of comments :

```
1216 | /* This function returns the position of the highest bit set in the lowest  

1217 |  * byte of <x>, between 0 and 7. It only works if <x> is non-null. It uses  

1218 |  * a 32-bit value as a lookup table to return one of 4 values for the  

1219 |  * highest 16 possible 4-bit values.  

1220 | */  

1221 | int flsnz8(unsigned int x)  

1222 | {  

1223 |     int ret = 0;  

1224 |     if (x >> 4) { x >=> 4; ret += 4; }  

1225 |     return ret + ((0xFFFFFAA50U >> (x << 1)) & 3) + 1;  

1226 | }  

1227 | ...  

1228 | bit = ~len + (skip << 3) + 9; /* (skip << 3) + (8 - len), saves 1 cycle */
```

1236 12) Use of assembly

1237 -----

1238 There are many projects where use of assembly code is not welcome. There is no
 1239 problem with use of assembly in haproxy, provided that :

- 1240 a) an alternate C-form is provided for architectures not covered
- 1241 b) the code is small enough and well commented enough to be maintained

1242 It is important to take care of various incompatibilities between compiler
 1243 versions, for instance regarding output and clobbered registers. There are
 1244 a number of documentations on the subject on the net. Anyway if you are
 1245 fiddling with assembly, you probably know that already.

1246 Example :

```
1247 | /* gcc does not know when it can safely divide 64 bits by 32 bits. Use this  

1248 |  * function when you know for sure that the result fits in 32 bits, because  

1249 |  * it is optimal on x86 and on 64bit processors.  

1250 |  */  

1251 | static inline unsigned int div64_32(unsigned long long o1, unsigned int o2)  

1252 | {  

1253 |     unsigned int result;  

1254 |     #ifdef __i386__  

1255 |         asm("divl %2"  

1256 |             : "=a" (result)  

1257 |             : "A"(o1), "rm"(o2));  

1258 |     #else  

1259 |         result = o1 / o2;  

1260 |     #endif  

1261 |     return result;  

1262 | }
```