```
2011/12/30 - HAProxy coding style - Willy Tarreau <w@1wt.eu>
------------------------------------------------------------

A number of contributors are often embarrassed with coding style issues, they
don't always know if they're doing it right, especially since the coding style
has elvoved along the years. What is explained here is not necessarily what is
applied in the code, but new code should as much as possible conform to this
style. Coding style fixes happen when code is replaced. It is useless to send
patches to fix coding style only, they will be rejected, unless they belong to
a patch series which needs these fixes prior to get code changes. Also, please
avoid fixing coding style in the same patches as functional changes, they make
code review harder.

When modifying a file, you must accept the terms of the license of this file
which is recalled at the top of the file, or is explained in the LICENSE file,
or if not stated, defaults to LGPL version 2.1 or later for files in the
'include' directory, and GPL version 2 or later for all other files.

When adding a new file, you must add a copyright banner at the top of the
file with your real name, e-mail address and a reminder of the license.
Contributions under incompatible licenses or too restrictive licenses might
get rejected. If in doubt, please apply the principle above for existing files.

All code examples below will intentionally be prefixed with "  | " to mark
where the code aligns with the first column, and tabs in this document will be
represented as a series of 8 spaces so that it displays the same everywhere.


1) Indentation and alignment
----------------------------

1.1) Indentation
----------------

Indentation and alignment are two completely different things that people often
get wrong. Indentation is used to mark a sub-level in the code. A sub-level
means that a block is executed in the context of another block (eg: a function
or a condition) :

  | main(int argc, char **argv)
  | {
  |         int i;
  |
  |         if (argc < 2)
  |                 exit(1);
  | }

In the example above, the code belongs to the main() function and the exit()
call belongs to the if statement. Indentation is made with tabs (\t, ASCII 9),
which allows any developer to configure their preferred editor to use their
own tab size and to still get the text properly indented. Exactly one tab is
used per sub-level. Tabs may only appear at the beginning of a line or after
another tab. It is illegal to put a tab after some text, as it mangles displays
in a different manner for different users (particularly when used to align
comments or values after a #define). If you're tempted to put a tab after some
text, then you're doing it wrong and you need alignment instead (see below).

Note that there are places where the code was not properly indented in the
past. In order to view it correctly, you may have to set your tab size to 8
characters.


1.2) Alignment
--------------
```



```
Alignment is used to continue a line in a way to makes things easier to group
together. By definition, alignment is character-based, so it uses spaces. Tabs
would not work because for one tab there would not be as many characters on all
displays. For instance, the arguments in a function declaration may be broken
into multiple lines using alignment spaces :

  | int http_header_match2(const char *hdr, const char *end,
  |                        const char *name, int len)
  | {
  | ...
  | }

In this example, the "const char *name" part is aligned with the first
character of the group it belongs to (list of function arguments). Placing it
here makes it obvious that it's one of the function's arguments. Multiple lines
are easy to handle this way. This is very common with long conditions too :

  |         if ((len < eol - sol) &&
  |             (sol[len] == ':') &&
  |             (strncasecmp(sol, name, len) == 0)) {
  |                 ctx->del = len;
  |         }

If we take again the example above marking tabs with "[-Tabs-]" and spaces
with "#", we get this :

  | [-Tabs-]if ((len < eol - sol) &&
  | [-Tabs-]####(sol[len] == ':') &&
  | [-Tabs-]####(strncasecmp(sol, name, len) == 0)) {
  | [-Tabs-][-Tabs-]ctx->del = len;
  | [-Tabs-]}

It is worth noting that some editors tend to confuse indentations and aligment.
Emacs is notoriously known for this brokenness, and is responsible for almost
all of the alignment mess. The reason is that Emacs only counts spaces, tries
to fill as many as possible with tabs and completes with spaces. Once you know
it, you just have to be careful, as alignment is not used much, so generally it
is just a matter of replacing the last tab with 8 spaces when this happens.

Indentation should be used everywhere there is a block or an opening brace. It
is not possible to have two consecutive closing braces on the same column, it
means that the innermost was not indented.

Right :

  | main(int argc, char **argv)
  | {
  |         if (argc > 1) {
  |                 printf("Hello\n");
  |         }
  |         exit(0);
  | }

Wrong :

  | main(int argc, char **argv)
  | {
  | if (argc > 1) {
  |         printf("Hello\n");
  | }
  | exit(0);
  | }

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 :

```
  | switch (*arg) {
  | case 'A': {
  |         int i;
  |         for (i = 0; i < 10; i++)
  |                 printf("Please stop pressing 'A'!\n");
  |         break;
  | }
  | case 'B':
  |         printf("You pressed 'B'\n");
  |         break;
  | case 'C':
  | case 'D':
  |         printf("You pressed 'C' or 'D'\n");
  |         break;
  | default:
  |         printf("I don't know what you pressed\n");
  | }
```

2) Braces

Braces are used to delimit multiple-instruction blocks. In general it is
preferred to avoid braces around single-instruction blocks as it reduces the
number of lines :

Right :

```
  | if (argc >= 2)
  |         exit(0);
```

Wrong :

```
  | if (argc >= 2) {
  |         exit(0);
  | }
```

But it is not that strict, it really depends on the context. It happens from
time to time that single-instruction blocks are enclosed within braces because
it makes the code more symmetrical, or more readable. Example :

```
  | if (argc < 2) {
  |         printf("Missing argument\n");
  |         exit(1);
  | } else {
  |         exit(0);
  | }
```

Braces are always needed to declare a function. A function's opening brace must
be placed at the beginning of the next line :

Right :

```
  | int main(int argc, char **argv)
  | {
  |         exit(0);
  | }
```

Wrong :


```
  | int main(int argc, char **argv) {
  |         exit(0);
  | }
```

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

Right :

```
  | int foo(const char *hdr, const char *end,
  |         const char *name, const char *err,
  |         int len)
  | {
  |         int i;
```

Wrong :

```
  | int foo(const char *hdr, const char *end,
  |         const char *name, const char *err,
  |         int len) {
  |         int i;
```

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 :

```
  | if (argc < 2)
  |         /* ret must not be negative here */
  |         if (ret < 0)
  |                 return -1;
```

Wrong change :

```
  | if (argc < 2)
  |         /* ret must not be negative here */
  |         if (ret < 0)
  |                 return -1;
  | else
  |         return 0;
```

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

```
  | if (argc < 2)
  |         /* ret must not be negative here */
  |         if (ret < 0)
  |                 return -1;
  |         else
  |                 return 0;
```

Right :

```
  | if (argc < 2) {
  |         /* ret must not be negative here */
  |         if (ret < 0)
  |                 return -1;
  | }
  | else
  |         return 0;
```

Similarly dangerous example :

```
  | if (ret < 0)
  |         /* ret must not be negative here */
  |         complain();
  | init();
```

Wrong change to silent the annoying message :

```
  | if (ret < 0)
  |         /* ret must not be negative here */
  |         //complain();
  | init();
```

... which in fact means :

```
  | if (ret < 0)
  |         init();
```

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 :

```
  | if (omult % idiv == 0) { omult /= idiv; idiv = 1; }
  | if (idiv % omult == 0) { idiv /= omult; omult = 1; }
  | if (imult % odiv == 0) { imult /= odiv; odiv = 1; }
  | if (odiv % imult == 0) { odiv /= imult; imult = 1; }
```

than in this one :

```
  | if (omult % idiv == 0) {
  |         omult /= idiv;
  |         idiv = 1;
  | }
  | if (idiv % omult == 0) {
  |         idiv /= omult;
  |         omult = 1;
  | }
  | if (imult % odiv == 0) {
  |         imult /= odiv;
  |         odiv = 1;
  | }
  | if (odiv % imult == 0) {
  |         odiv /= imult;
  |         imult = 1;
  | }
```

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 :


```
  | switch (*arg) {
  | case 'A': ret = 1; break;
  | case 'B': ret = 2; break;
  | case 'C': ret = 4; break;
  | case 'D': ret = 8; break;
  | default : ret = 0; break;
  | }
```

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 :

```
  | switch (*arg) {
  | case 'A':
  |         if (ret < 0)
  |                 ret = 1;
  |         break;
  | default : ret = 0; break;
  | }
```

Wrong :

```
  | switch (*arg) {
  | case 'A': if (ret < 0)
  |                 ret = 1;
  |         break;
  | default : ret = 0; break;
  | }
```

Right :

```
  | if (argc < 2)
  |         if (ret < 0)
  |                 return -1;
```

or Right :

```
  | if (argc < 2)
  |         if (ret < 0) return -1;
```

but Wrong :

```
  | if (argc < 2) if (ret < 0) return -1;
```

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

Right :

```
  | if ((txn->flags & TX_NOT_FIRST) &&
  |     ((req->flags & BF_FULL) ||
  |      req->r < req->lr ||
  |      req->r > req->data + req->size - global.tune.maxrewrite)) {
  |         return 0;
  | }
```

Right :

```
391
392     | if ((txn->flags & TX_NOT_FIRST)
393     |     && ((req->flags & BF_FULL)
394     |          || req->r < req->lr
395     |          || req->r > req->data + req->size - global.tune.maxrewrite)) {
396     |          return 0;
397     | }
398
399   Wrong :
400
401     | if ((txn->flags & TX_NOT_FIRST) &&
402     |    ((req->flags & BF_FULL) ||
403     |      req->r < req->lr
404     |    || req->r > req->data + req->size - global.tune.maxrewrite)) {
405     |          return 0;
406     | }
407
408   If it makes the result more readable, parenthesis may even be closed on their
409   own line in order to align with the opening one. Note that should normally not
410   be needed because such code would be too complex to be digged into.
411
412   The "else" statement may either be merged with the closing "if" brace or lie on
413   its own line. The later is preferred but it adds one extra line to each control
414   block which is annoying in short ones. However, if the "else" is followed by an
415   "if", then it should really be on its own line and the rest of the if/else
416   blocks must follow the same style.
417
418   Right :
419
420     | if (a < b) {
421     |          return a;
422     | }
423     | else {
424     |          return b;
425     | }
426
427   Right :
428
429     | if (a < b) {
430     |          return a;
431     | } else {
432     |          return b;
433     | }
434
435   Right :
436
437     | if (a < b) {
438     |          return a;
439     | }
440     | else if (a != b) {
441     |          return b;
442     | }
443     | else {
444     |          return 0;
445     | }
446
447   Wrong :
448
449     | if (a < b) {
450     |          return a;
451     | } else if (a != b) {
452     |          return b;
453     | } else {
454     |          return 0;
455     | }
```



```
456
457   Wrong :
458
459     | if (a < b) {
460     |          return a;
461     | }
462     | else if (a != b) {
463     |          return b;
464     | } else {
465     |          return 0;
466     | }
467
468
469   4) Spacing
470   ----------
471
472   Correctly spacing code is very important. When you have to spot a bug at 3am,
473   you need it to be clear. When you expect other people to review your code, you
474   want it to be clear and don't want them to get nervous when trying to find what
475   you did.
476
477   Always place spaces around all binary or ternary operators, commas, as well as
478   after semi-colons and opening braces if the line continues :
479
480   Right :
481
482     | int ret = 0;
483     | /* if (x >> 4) { x >>= 4; ret += 4; } */
484     | ret += (x >> 4) ? (x >>= 4, 4) : 0;
485     | val = ret + ((0xFFFFAA50U >> (x << 1)) & 3) + 1;
486
487   Wrong :
488
489     | int ret=0;
490     | /* if (x>>4) {x>>=4;ret+=4;} */
491     | ret+=(x>>4)?(x>>=4,4):0;
492     | val=ret+((0xFFFFAA50U>>(x<<1))&3)+1;
493
494   Never place spaces after unary operators (&, *, -, !, ~, ++, --) nor cast, as
495   they might be confused with they binary counterpart, nor before commas or
496   semicolons :
497
498   Right :
499
500     | bit = !!(~len++ ^ -(unsigned char)*x);
501
502   Wrong :
503
504     | bit = ! ! (~len++ ^ - (unsigned char) * x) ;
505
506   Note that "sizeof" is a unary operator which is sometimes considered as a
507   langage keyword, but in no case it is a function. It does not require
508   parenthesis so it is sometimes followed by spaces and sometimes not when
509   there are no parenthesis. Most people do not really care as long as what
510   is written is unambiguous.
511
512   Braces opening a block must be preceeded by one space unless the brace is
513   placed on the first column :
514
515   Right :
516
517     | if (argc < 2) {
518     | }
519
520   Wrong :
```

```
   | if (argc < 2){
   | }
```

Do not add unneeded spaces inside parenthesis, they just make the code less
readable.

Right :

```
   | if (x < 4 && (!y || !z))
   |         break;
```

Wrong :

```
   | if ( x < 4 && ( !y || !z ) )
   |         break;
```

Language keywords must all be followed by a space. This is true for control
statements (do, for, while, if, else, return, switch, case), and for types
(int, char, unsigned). As an exception, the last type in a cast does not take
a space before the closing parenthesis). The "default" statement in a "switch"
construct is generally just followed by the colon. However the colon after a
"case" or "default" statement must be followed by a space.

Right :

```
   | if (nbargs < 2) {
   |         printf("Missing arg at %c\n", *(char *)ptr);
   |         for (i = 0; i < 10; i++) beep();
   |         return 0;
   | }
   | switch (*arg) {
```

Wrong :

```
   | if(nbargs < 2){
   |         printf("Missing arg at %c\n", *(char*)ptr);
   |         for(i = 0; i < 10; i++)beep();
   |         return 0;
   | }
   | switch(*arg) {
```

Function calls are different, the opening parenthesis is always coupled to the
function name without any space. But spaces are still needed after commas :

Right :

```
   | if (!init(argc, argv))
   |         exit(1);
```

Wrong :

```
   | if (!init (argc,argv))
   |         exit(1);
```

5) Excess or lack of parenthesis

Sometimes there are too many parenthesis in some formulas, sometimes there are
too few. There are a few rules of thumb for this. The first one is to respect
the compiler's advice. If it emits a warning and asks for more parenthesis to
avoid confusion, follow the advice at least to shut the warning. For instance,
the code below is quite ambiguous due to its alignment :


```
   | if (var1 < 2 || var2 < 2 &&
   |     var3 != var4) {
   |         /* fail */
   |         return -3;
   | }
```

Note that this code does :

```
   | if (var1 < 2 || (var2 < 2 && var3 != var4)) {
   |         /* fail */
   |         return -3;
   | }
```

But maybe the author meant :

```
   | if ((var1 < 2 || var2 < 2) && var3 != var4) {
   |         /* fail */
   |         return -3;
   | }
```

A second rule to put parenthesis is that people don't always know operators
precedence too well. Most often they have no issue with operators of the same
category (eg: booleans, integers, bit manipulation, assignment) but once these
operators are mixed, it causes them all sort of issues. In this case, it is
wise to use parenthesis to avoid errors. One common error concerns the bit
shift operators because they're used to replace multiplies and divides but
don't have the same precedence :

The expression :

```
   | x = y * 16 + 5;
```

becomes :

```
   | x = y << 4 + 5;
```

which is wrong because it is equivalent to :

```
   | x = y << (4 + 5);
```

while the following was desired instead :

```
   | x = (y << 4) + 5;
```

It is generally fine to write boolean expressions based on comparisons without
any parenthesis. But on top of that, integer expressions and assignments should
then be protected. For instance, there is an error in the expression below
which should be safely rewritten :

Wrong :

```
   | if (var1 > 2 && var1 < 10 ||
   |     var1 > 2 + 256 && var2 < 10 + 256 ||
   |     var1 > 2 + 1 << 16 && var2 < 10 + 2 << 16)
   |         return 1;
```

Right (may remove a few parenthesis depending on taste) :

```
   | if ((var1 > 2 && var1 < 10) ||
   |     (var1 > (2 + 256) && var2 < (10 + 256)) ||
   |     (var1 > (2 + (1 << 16)) && var2 < (10 + (1 << 16))))
   |         return 1;
```

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 :

Wrong :

```
| int ret0()
| {
|         return(0);
| }
```

Right :

```
| int ret0()
| {
|         return 0;
| }
```

Parenthesisis 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.

Wrong :

```
| void *check_private_data(void *arg1, void *arg2)
| {
|         char *area;
|
|         if (*(int *)arg1 > 1000)
|                 return NULL;
|         if (memcmp(*(const char *)arg2, "send(", 5) != 0))
|                 return NULL;
|         area = malloc(*(int *)arg1);
|         if (!area)
|                 return NULL;
|         memcpy(area, *(const char *)arg2 + 5, *(int *)arg1);
|         return area;
| }
```

Right :

```
| void *check_private_data(void *arg1, void *arg2)
| {
|         char *area;
|         int len = *(int *)arg1;
|         const char *msg = arg2;
|
|         if (len > 1000)
|                 return NULL;
|         if (memcmp(msg, "send(", 5) != 0)
|                 return NULL;
|         area = malloc(len);
|         if (!area)
|                 return NULL;
|         memcpy(area, msg + 5, len);
|         return area;
| }
```

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 :

```
| if (!isdigit(*c) && !isspace(*c))
|         break;
```

is easier to understand than :

```
| if (isdigit(*c) == 0 && isspace(*c) == 0)
|         break;
```

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 :

```
| void my_strcpy(char *d, const char *s)
| {
|         while ((*d++ = *s++));
| }
```

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 :

```
| area = malloc(1000);
| if (!area)
|         return -1;
```

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) :

```
    strcmp(a, b) == 0  <=>  a == b
    strcmp(a, b) != 0  <=>  a != b
    strcmp(a, b) <  0  <=>  a <  b
    strcmp(a, b) >  0  <=>  a >  b
```

Avoid this :

```
| if (strcmp(arg, "test"))
|         printf("this is not a test\n");
|
| if (!strcmp(arg, "test"))
|         printf("this is a test\n");
```

Prefer this :

```
| if (strcmp(arg, "test") != 0)
|         printf("this is not a test\n");
```

```
  |
  | if (strcmp(arg, "test") == 0)
  |         printf("this is a test\n");
```

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 :

```
  | if ((fd = open(file, O_RDONLY)) < 0)
  |         return -1;
```

This is wrong. The man page says that -1 is returned if an error occured. 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, eventhough 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 :

```
  | int delay_expired(timer_t exp, timer_us_t now)
  | {
  |         return now >= exp;
  | }
```

With the types declared in another file this way :

```
  | typedef unsigned int timer_t;
  | typedef struct timeval timer_us_t;
```

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 :

```
  | int delay_expired(unsigned int exp, struct timeval *now)
  | {
  |         return now >= exp->tv_sec;
  | }
```

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 preceeded by a comment :

```
  | enum {
  |         first = 0,
  |         second,
  |         third,
```



```
  |         fourth,
  | };


  | enum {
  |         first = 0,
  |         second,
  |         third,
  |         fourth,
  |         /* nbvalues must always be placed last */
  |         nbvalues
  | };
```

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

Wrong :

```
  | struct request_args { /* arguments on the query string */
  |         char *name;
  |         char *value;
  |         struct misc_args *next;
  | };
```

Right :

```
  | struct qs_args { /* arguments on the query string */
  |         char *name;
  |         char *value;
  |         struct qs_args *next;
  | }
```

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 :

```
  - generateTcpipIsn()
  - generateTcpIpIsn()
  - generateTcpIpISN()
  - generateTCPIPISN()
  etc...
```

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 short names.

Valid examples :

```
  - generate_tcpip_isn()
  - generate_tcp_ip_isn()
  - generate_TCPIP_ISN()
  - generate_TCP_IP_ISN()
```

Another example is easy to understand when 3 arguments are involved in naming the function :

Wrong (naming conflict) :

```
  | /* returns A + B * C */
  | int mulABC(int a, int b, int c)
```

```
   | {
   |         return a + b * c;
   | }
   |
   | /* returns (A + B) * C */
   | int mulABC(int a, int b, int c)
   | {
   |         return (a + b) * c;
   | }
```

Right (unambiguous naming) :

```
   | /* returns A + B * C */
   | int mul_a_bc(int a, int b, int c)
   | {
   |         return a + b * c;
   | }
   |
   | /* returns (A + B) * C */
   | int mul_ab_c(int a, int b, int c)
   | {
   |         return (a + b) * c;
   | }
```

Whenever you manipulate pointers, try to declare them as "const", as it will save you from many accidental misuses and will only cause warnings to be emitted when there is a real risk. In the examples below, it is possible to call my_strcpy() with a const string only in the first declaration. Note that people who ignore "const" are often the ones who cast a lot and who complain from segfaults when using strtok() !

Right :

```
   | void my_strcpy(char *d, const char *s)
   | {
   |         while ((*d++ = *s++));
   | }
   |
   | void say_hello(char *dest)
   | {
   |         my_strcpy(dest, "hello\n");
   | }
```

Wrong :

```
   | void my_strcpy(char *d, char *s)
   | {
   |         while ((*d++ = *s++));
   | }
   |
   | void say_hello(char *dest)
   | {
   |         my_strcpy(dest, "hello\n");
   | }
```

9) Getting macros right

It is very common for macros to do the wrong thing when used in a way their author did not have in mind. For this reason, macros must always be named with uppercase letters only. This is the only way to catch the developer's eye when using them, so that he double-checks whether he's taking risks or not. First, macros must never ever be terminated by a semi-colon, or they will close the wrong block once in a while. For instance, the following will cause a build

error before the "else" due to the double semi-colon :

Wrong :

```
   | #define WARN printf("warning\n");
   | ...
   |         if (a < 0)
   |                 WARN;
   |         else
   |                 a--;
```

Right :

```
   | #define WARN printf("warning\n")
```

If multiple instructions are needed, then use a do { } while (0) block, which is the only construct which respects *exactly* the semantics of a single instruction :

```
   | #define WARN do { printf("warning\n"); log("warning\n"); } while (0)
   | ...
   |
   |         if (a < 0)
   |                 WARN;
   |         else
   |                 a--;
```

Second, do not put unprotected control statements in macros, they will definitely cause bugs :

Wrong :

```
   | #define WARN if (verbose) printf("warning\n")
   | ...
   |         if (a < 0)
   |                 WARN;
   |         else
   |                 a--;
```

Which is equivalent to the undesired form below :

```
   |         if (a < 0)
   |                 if (verbose)
   |                         printf("warning\n");
   |                 else
   |                         a--;
```

Right way to do it :

```
   | #define WARN do { if (verbose) printf("warning\n"); } while (0)
   | ...
   |         if (a < 0)
   |                 WARN;
   |         else
   |                 a--;
```

Which is equivalent to :

```
   |         if (a < 0)
   |                 do { if (verbose) printf("warning\n"); } while (0);
   |         else
   |                 a--;
```

Macro parameters must always be surrounded by parenthesis, and must never be duplicated in the same macro unless explicitly stated. Also, macros must not be

defined with operators without surrounding parenthesis. The MIN/MAX macros are
a pretty common example of multiple misuses, but this happens as early as when
using bit masks. Most often, in case of any doubt, try to use inline functions
instead.

Wrong :

```
  | #define MIN(a, b) a < b ? a : b
  |
  | /* returns 2 * min(a,b) + 1 */
  | int double_min_p1(int a, int b)
  | {
  |         return 2 * MIN(a, b) + 1;
  | }
```

What this will do :

```
  | int double_min_p1(int a, int b)
  | {
  |         return 2 * a < b ? a : b + 1;
  | }
```

Which is equivalent to :

```
  | int double_min_p1(int a, int b)
  | {
  |         return (2 * a) < b ? a : (b + 1);
  | }
```

The first thing to fix is to surround the macro definition with parenthesis to
avoid this mistake :

```
  | #define MIN(a, b) (a < b ? a : b)
```

But this is still not enough, as can be seen in this example :

```
  | /* compares either a or b with c */
  | int min_ab_c(int a, int b, int c)
  | {
  |         return MIN(a ? a : b, c);
  | }
```

Which is equivalent to :

```
  | int min_ab_c(int a, int b, int c)
  | {
  |         return (a ? a : b < c ? a ? a : b : c);
  | }
```

Which in turn means a totally different thing due to precedence :

```
  | int min_ab_c(int a, int b, int c)
  | {
  |         return (a ? a : ((b < c) ? (a ? a : b) : c));
  | }
```

This can be fixed by surrounding *each* argument in the macro with parenthesis:

```
  | #define MIN(a, b) ((a) < (b) ? (a) : (b))
```

But this is still not enough, as can be seen in this example :

```
  | int min_ap1_b(int a, int b)
  | {
  |         return MIN(++a, b);
```



```
  | }
```

Which is equivalent to :

```
  | int min_ap1_b(int a, int b)
  | {
  |         return ((++a) < (b) ? (++a) : (b));
  | }
```

Again, this is wrong because "a" is incremented twice if below b. The only way
to fix this is to use a compound statement and to assign each argument exactly
once to a local variable of the same type :

```
  | #define MIN(a, b) ({ typeof(a) __a = (a); typeof(b) __b = (b);  \
  |                      ((__a) < (__b) ? (__a) : (__b));           \
  |                   })
```

At this point, using static inline functions is much cleaner if a single type
is to be used :

```
  | static inline int min(int a, int b)
  | {
  |         return a < b ? a : b;
  | }
```

10) Includes

Includes are as much as possible listed in alphabetically ordered groups :
  - the libc-standard includes (those without any path component)
  - the includes more or less system-specific (sys/*, netinet/*, ...)
  - includes from the local "common" subdirectory
  - includes from the local "types" subdirectory
  - includes from the local "proto" subdirectory

Each section is just visually delimited from the other ones using an empty
line. The two first ones above may be merged into a single section depending on
developer's preference. Please do not copy-paste include statements from other
files. Having too many includes significantly increases build time and makes it
hard to find which ones are needed later. Just include what you need and if
possible in alphabetical order so that when something is missing, it becomes
obvious where to look for it and where to add it.

All files should include <common/config.h> because this is where build options
are prepared.

Header files are split in two directories ("types" and "proto") depending on
what they provide. Types, structures, enums and #defines must go into the
"types" directory. Function prototypes and inlined functions must go into the
"proto" directory. This split is because of inlined functions which
cross-reference types from other files, which cause a chicken-and-egg problem
if the functions and types are declared at the same place.

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

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

11) Comments

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

```
  | /*
  |  * This is a multi-line
  |  * comment.
  |  */
```

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

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

Wrong use of comments :

```
  | int flsnz8(unsigned int x)
  | {
  |         int ret = 0;                          /* initialize ret */
  |         if (x >> 4) { x >>= 4; ret += 4; }    /* add 4 to ret if needed */
  |         return ret + ((0xFFFFAA50U >> (x << 1)) & 3) + 1; /* add ??? */
  | }
  | ...
  | bit = ~len + (skip << 3) + 9;        /* update bit */
```

Right use of comments :

```
  | /* This function returns the positoin of the highest bit set in the lowest
  |  * byte of <x>, between 0 and 7. It only works if <x> is non-null. It uses
  |  * a 32-bit value as a lookup table to return one of 4 values for the
  |  * highest 16 possible 4-bit values.
  |  */
  | int flsnz8(unsigned int x)
  | {
  |         int ret = 0;
  |         if (x >> 4) { x >>= 4; ret += 4; }
  |         return ret + ((0xFFFFAA50U >> (x << 1)) & 3) + 1;
  | }
  | ...
  | bit = ~len + (skip << 3) + 9; /* (skip << 3) + (8 - len), saves 1 cycle */
```

12) Use of assembly

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

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

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

Example :

```
  | /* gcc does not know when it can safely divide 64 bits by 32 bits. Use this
  |  * function when you know for sure that the result fits in 32 bits, because
  |  * it is optimal on x86 and on 64bit processors.
  |  */
  | static inline unsigned int div64_32(unsigned long long o1, unsigned int o2)
  | {
  |         unsigned int result;
  | #ifdef __i386__
  |         asm("divl %2"
  |             : "=a" (result)
  |             : "A"(o1), "rm"(o2));
  | #else
  |         result = o1 / o2;
  | #endif
  |         return result;
  | }
```