```
**********************************************************
   25390 Mon Jan  5 17:57:22 2015
new/usr/src/tools/scripts/cstyle.pl
5506 cstyle: warn about #if enumerating ISA defines
**********************************************************
```

```
   1 #!/usr/bin/perl -w
   2 #
   3 # CDDL HEADER START
   4 #
   5 # The contents of this file are subject to the terms of the
   6 # Common Development and Distribution License (the "License").
   7 # You may not use this file except in compliance with the License.
   8 #
   9 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10 # or http://www.opensolaris.org/os/licensing.
  11 # See the License for the specific language governing permissions
  12 # and limitations under the License.
  13 #
  14 # When distributing Covered Code, include this CDDL HEADER in each
  15 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16 # If applicable, add the following below this CDDL HEADER, with the
  17 # fields enclosed by brackets "[]" replaced with your own identifying
  18 # information: Portions Copyright [yyyy] [name of copyright owner]
  19 #
  20 # CDDL HEADER END
  21 #
  22 #
  23 # Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
  24 # Use is subject to license terms.
  25 # Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
  25 #
  26 # @(#)cstyle 1.58 98/09/09 (from shannon)
  27 #ident  "%Z%%M% %I%     %E% SMI"
  26 #
  27 # cstyle - check for some common stylistic errors.
  28 #
  29 #       cstyle is a sort of "lint" for C coding style.
  30 #       It attempts to check for the style used in the
  31 #       kernel, sometimes known as "Bill Joy Normal Form".
  32 #
  33 #       There's a lot this can't check for, like proper indentation
  34 #       of code blocks.  There's also a lot more this could check for.
  35 #
  36 #       A note to the non perl literate:
  37 #
  38 #               perl regular expressions are pretty much like egrep
  39 #               regular expressions, with the following special symbols
  40 #
  41 #               \s      any space character
  42 #               \S      any non-space character
  43 #               \w      any "word" character [a-zA-Z0-9_]
  44 #               \W      any non-word character
  45 #               \d      a digit [0-9]
  46 #               \D      a non-digit
  47 #               \b      word boundary (between \w and \W)
  48 #               \B      non-word boundary
  49 #

  51 require 5.0;
  52 use IO::File;
  53 use Getopt::Std;
  54 use strict;

  56 my $usage =
  57 "usage: cstyle [-chpvCP] [-o constructs] file ...
  58         -c      check continuation indentation inside functions
```



```
  59         -h      perform heuristic checks that are sometimes wrong
  60         -p      perform some of the more picky checks
  61         -v      verbose
  62         -C      don't check anything in header block comments
  63         -P      check for use of non-POSIX types
  64         -o constructs
  65                 allow a comma-seperated list of optional constructs:
  66                     doxygen     allow doxygen-style block comments (/** /*!)
  67                     splint      allow splint-style lint comments (/*@ ... @*/)
  68 ";

  70 my %opts;

  72 if (!getopts("cho:pvCP", \%opts)) {
  73         print $usage;
  74         exit 2;
  75 }
_____unchanged_portion_omitted_

 214 sub cstyle($$) {

 216 my ($fn, $filehandle) = @_;
 217 $filename = $fn;                        # share it globally

 219 my $in_cpp = 0;
 220 my $next_in_cpp = 0;

 222 my $in_comment = 0;
 223 my $in_header_comment = 0;
 224 my $comment_done = 0;
 225 my $in_warlock_comment = 0;
 226 my $in_function = 0;
 227 my $in_function_header = 0;
 228 my $in_declaration = 0;
 229 my $note_level = 0;
 230 my $nextok = 0;
 231 my $nocheck = 0;

 233 my $in_string = 0;

 235 my ($okmsg, $comment_prefix);

 237 $line = '';
 238 $prev = '';
 239 reset_indent();

 241 line: while (<$filehandle>) {
 242         s/\r?\n$//;     # strip return and newline

 244         # save the original line, then remove all text from within
 245         # double or single quotes, we do not want to check such text.

 247         $line = $_;

 249         #
 250         # C allows strings to be continued with a backslash at the end of
 251         # the line.  We translate that into a quoted string on the previous
 252         # line followed by an initial quote on the next line.
 253         #
 254         # (we assume that no-one will use backslash-continuation with character
 255         # constants)
 256         #
 257         $_ = '"' . $_           if ($in_string && !$nocheck && !$in_comment);

 259         #
 260         # normal strings and characters
```

```
261         #
262         s/'([^\\']|\\[^xX0]|\\0[0-9]*|\\[xX][0-9a-fA-F]*)'/''/g;
263         s/"([^\\"]|\\.)*"/\"\"/g;
264
265         #
266         # detect string continuation
267         #
268         if ($nocheck || $in_comment) {
269                 $in_string = 0;
270         } else {
271                 #
272                 # Now that all full strings are replaced with "", we check
273                 # for unfinished strings continuing onto the next line.
274                 #
275                 $in_string =
276                     (s/([^"](?:"")*)"([^\\"]|\\.)*\\$/$1""/ ||
277                     s/^("")*"([^\\"]|\\.)*\\$/""/);
278         }
279
280         #
281         # figure out if we are in a cpp directive
282         #
283         $in_cpp = $next_in_cpp || /^\s*#/;      # continued or started
284         $next_in_cpp = $in_cpp && /\\$/;        # only if continued
285
286         # strip off trailing backslashes, which appear in long macros
287         s/\s*\\$//;
288
289         # an /* END CSTYLED */ comment ends a no-check block.
290         if ($nocheck) {
291                 if (/\/\* *END *CSTYLED *\*\//) {
292                         $nocheck = 0;
293                 } else {
294                         reset_indent();
295                         next line;
296                 }
297         }
298
299         # a /*CSTYLED*/ comment indicates that the next line is ok.
300         if ($nextok) {
301                 if ($okmsg) {
302                         err($okmsg);
303                 }
304                 $nextok = 0;
305                 $okmsg = 0;
306                 if (/\/\* *CSTYLED.*\*\//) {
307                         /^.*\/\* *CSTYLED *(.*) *\*\/.*$/;
308                         $okmsg = $1;
309                         $nextok = 1;
310                 }
311                 $no_errs = 1;
312         } elsif ($no_errs) {
313                 $no_errs = 0;
314         }
315
316         # check length of line.
317         # first, a quick check to see if there is any chance of being too long.
318         if (($line =~ tr/\t/\t/) * 7 + length($line) > 80) {
319                 # yes, there is a chance.
320                 # replace tabs with spaces and check again.
321                 my $eline = $line;
322                 1 while $eline =~
323                     s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e;
324                 if (length($eline) > 80) {
325                         err("line > 80 characters");
326                 }
```



```
327         }
328
329         # ignore NOTE(...) annotations (assumes NOTE is on lines by itself).
330         if ($note_level || /\b_?NOTE\s*\(/) { # if in NOTE or this is NOTE
331                 s/[^()]//g;                     # eliminate all non-parens
332                 $note_level += s/\(//g - length;  # update paren nest level
333                 next;
334         }
335
336         # a /* BEGIN CSTYLED */ comment starts a no-check block.
337         if (/\/\* *BEGIN *CSTYLED *\*\//) {
338                 $nocheck = 1;
339         }
340
341         # a /*CSTYLED*/ comment indicates that the next line is ok.
342         if (/\/\* *CSTYLED.*\*\//) {
343                 /^.*\/\* *CSTYLED *(.*) *\*\/.*$/;
344                 $okmsg = $1;
345                 $nextok = 1;
346         }
347         if (/\/\/ *CSTYLED/) {
348                 /^.*\/\/ *CSTYLED *(.*)$/;
349                 $okmsg = $1;
350                 $nextok = 1;
351         }
352
353         # universal checks; apply to everything
354         if (/\t +\t/) {
355                 err("spaces between tabs");
356         }
357         if (/ \t+ /) {
358                 err("tabs between spaces");
359         }
360         if (/\s$/) {
361                 err("space or tab at end of line");
362         }
363         if (/[^ \t(]\/\*/ && !/\w\(\/\*.*\*\/\);/) {
364                 err("comment preceded by non-blank");
365         }
366
367         # is this the beginning or ending of a function?
368         # (not if "struct foo\n{\n")
369         if (/^{$/ && $prev =~ /\)\s*(const\s*)?(\/\*.*\*\/\s*)?\\?$/) {
370                 $in_function = 1;
371                 $in_declaration = 1;
372                 $in_function_header = 0;
373                 $prev = $line;
374                 next line;
375         }
376         if (/^}\s*(\/\*.*\*\/\s*)*$/) {
377                 if ($prev =~ /^\s*return\s*;/) {
378                         err_prev("unneeded return at end of function");
379                 }
380                 $in_function = 0;
381                 reset_indent();         # we don't check between functions
382                 $prev = $line;
383                 next line;
384         }
385         if (/^\w*\($/) {
386                 $in_function_header = 1;
387         }
388
389         if ($in_warlock_comment && /\*\//) {
390                 $in_warlock_comment = 0;
391                 $prev = $line;
392                 next line;
```

```
393         }
394
395         # a blank line terminates the declarations within a function.
396         # XXX - but still a problem in sub-blocks.
397         if ($in_declaration && /^$/) {
398                 $in_declaration = 0;
399         }
400
401         if ($comment_done) {
402                 $in_comment = 0;
403                 $in_header_comment = 0;
404                 $comment_done = 0;
405         }
406         # does this looks like the start of a block comment?
407         if (/$hdr_comment_start/) {
408                 if (!/^\t*\/\*/) {
409                         err("block comment not indented by tabs");
410                 }
411                 $in_comment = 1;
412                 /^(\s*)\//;
413                 $comment_prefix = $1;
414                 if ($comment_prefix eq "") {
415                         $in_header_comment = 1;
416                 }
417                 $prev = $line;
418                 next line;
419         }
420         # are we still in the block comment?
421         if ($in_comment) {
422                 if (/^$comment_prefix \*\/$/) {
423                         $comment_done = 1;
424                 } elsif (/\*\//) {
425                         $comment_done = 1;
426                         err("improper block comment close")
427                             unless ($ignore_hdr_comment && $in_header_comment);
428                 } elsif (!/^$comment_prefix \*[ \t]/ &&
429                     !/^$comment_prefix \*$/) {
430                         err("improper block comment")
431                             unless ($ignore_hdr_comment && $in_header_comment);
432                 }
433         }
434
435         if ($in_header_comment && $ignore_hdr_comment) {
436                 $prev = $line;
437                 next line;
438         }
439
440         # check for errors that might occur in comments and in code.
441
442         # allow spaces to be used to draw pictures in header comments.
443         if (/[^ ]     / && !/".*     .*"/ && !$in_header_comment) {
444                 err("spaces instead of tabs");
445         }
446         if (/^ / && !/^ \*[ \t\/]/ && !/^ \*$/ &&
447             (!/^    \w/ || $in_function != 0)) {
448                 err("indent by spaces instead of tabs");
449         }
450         if (/^\t+ [^ \t\*]/ || /^\t+  \S/ || /^\t+   \S/) {
451                 err("continuation line not indented by 4 spaces");
452         }
453         if (/$warlock_re/ && !/\*\//) {
454                 $in_warlock_comment = 1;
455                 $prev = $line;
456                 next line;
457         }
458         if (/^\s*\/\*./ && !/^\s*\/\*.*\*\// && !/$hdr_comment_start/) {
```



```
459                 err("improper first line of block comment");
460         }
461
462         if ($in_comment) {      # still in comment, don't do further checks
463                 $prev = $line;
464                 next line;
465         }
466
467         if ((/[^(]\/\*\S/ || /^\/\*\S/) &&
468             !(/$lint_re/ || ($splint_comments && /$splint_re/))) {
469                 err("missing blank after open comment");
470         }
471         if (/\S\*\/[^)]|\S\*\/$/ &&
472             !(/$lint_re/ || ($splint_comments && /$splint_re/))) {
473                 err("missing blank before close comment");
474         }
475         if (/\/\/\S/) {         # C++ comments
476                 err("missing blank after start comment");
477         }
478         # check for unterminated single line comments, but allow them when
479         # they are used to comment out the argument list of a function
480         # declaration.
481         if (/\S.*\/\*/ && !/\S.*\/\*.*\*\// && !/\(\/\*/) {
482                 err("unterminated single line comment");
483         }
484
485         # check that #if and #elif don't enumerate ISA defines when there
486         # are more concise ways of checking.  E.g., don't do:
487         #     #if defined(__amd64) || defined(__i386)
488         # when there is:
489         #     #ifdef __x86
490         if ((/^(#if|#elif)\sdefined\((.*)\)\s\|\|\sdefined\((.*)\)/) ||
491             (/^(#if|#elif)\s!defined\((.*)\)\s&&\s!defined\((.*)\)/)) {
492                 my $directive = $1;
493                 my $first = $2;
494                 my $second = $3;
495                 ($first, $second) = ($second, $first) if ($first gt $second);
496
497                 if (($first eq "__amd64") && ($second eq "__i386")) {
498                         err("$directive checking for $first or $second " .
499                             "instead of __x86");
500                 }
501 #endif /* ! codereview */
502         }
503
504         if (/^(#else|#endif|#include)(.*)$/) {
505                 $prev = $line;
506                 if ($picky) {
507                         my $directive = $1;
508                         my $clause = $2;
509                         # Enforce ANSI rules for #else and #endif: no noncomment
510                         # identifiers are allowed after #endif or #else.  Allow
511                         # C++ comments since they seem to be a fact of life.
512                         if ((($1 eq "#endif") || ($1 eq "#else")) &&
513                             ($clause ne "") &&
514                             (!($clause =~ /^\s+\/\*.*\*\/$/)) &&
515                             (!($clause =~ /^\s+\/\/.*$/))) {
516                                 err("non-comment text following " .
517                                     "$directive (or malformed $directive " .
518                                     "directive)");
519                         }
520                 }
521                 next line;
522         }
523
524         #
```

```
525         # delete any comments and check everything else.  Note that
526         # ".*?" is a non-greedy match, so that we don't get confused by
527         # multiple comments on the same line.
528         #
529         s/\/\*.*?\*\//^A/g;
530         s/\/\/.*$/^A/;          # C++ comments

532         # delete any trailing whitespace; we have already checked for that.
533         s/\s*$//;

535         # following checks do not apply to text in comments.

537         if (/[^<>\s][!<>=]=/ || /[^<>][!<>=]=[^\s,]/ ||
538             (/[^->]>[^,=>\s]/ && !/[^->]>$/) ||
539             (/[^<]<[^,=<\s]/ && !/[^<]<$/) ||
540             /[^<\s]<[^<]/ || /[^->\s]>[^>]/) {
541                 err("missing space around relational operator");
542         }
543         if (/\S>>=/ || /\S<<=/ || />>=\S/ || /<<=\S/ || /\S[-+*\/&|^%]=/ ||
544             (/[^-+*\/&|^%!<>=\s]=[^=]/ && !/[^-+*\/&|^%!<>=\s]=$/) ||
545             (/[^!<>=]=[^=\s]/ && !/[^!<>=]=$/)) {
546                 # XXX - should only check this for C++ code
547                 # XXX - there are probably other forms that should be allowed
548                 if (!/\soperator=/) {
549                         err("missing space around assignment operator");
550                 }
551         }
552         if (/[,;]\S/ && !/\bfor \(;;\)/) {
553                 err("comma or semicolon followed by non-blank");
554         }
555         # allow "for" statements to have empty "while" clauses
556         if (/\s[,;]/ && !/^[\t]+;$/ && !/^\s*for \([^;]*; ;[^;]*\)/) {
557                 err("comma or semicolon preceded by blank");
558         }
559         if (/^\s*(&&|\|\|)/) {
560                 err("improper boolean continuation");
561         }
562         if (/\S   *(&&|\|\|)/ || /(&&|\|\|)   *\S/) {
563                 err("more than one space around boolean operator");
564         }
565         if (/\b(for|if|while|switch|sizeof|return|case)\(/) {
566                 err("missing space between keyword and paren");
567         }
568         if (/(\b(for|if|while|switch|return)\b.*){2,}/ && !/^#define/) {
569                 # multiple "case" and "sizeof" allowed
570                 err("more than one keyword on line");
571         }
572         if (/\b(for|if|while|switch|sizeof|return|case)\s\s+\(/ &&
573             !/^#if\s+\(/) {
574                 err("extra space between keyword and paren");
575         }
576         # try to detect "func (x)" but not "if (x)" or
577         # "#define foo (x)" or "int (*func)();"
578         if (/\w\s\(/) {
579                 my $s = $_;
580                 # strip off all keywords on the line
581                 s/\b(for|if|while|switch|return|case|sizeof)\s\(/XXX(/g;
582                 s/#elif\s\(/XXX(/g;
583                 s/^#define\s+\w+\s+\(/XXX(/;
584                 # do not match things like "void (*f)();"
585                 # or "typedef void (func_t)();"
586                 s/\w\s\(+\*/XXX(*/g;
587                 s/\b($typename|void)\s+\(+/XXX(/og;
588                 if (/\w\s\(/) {
589                         err("extra space between function name and left paren");
590                 }
```



```
591                 $_ = $s;
592         }
593         # try to detect "int foo(x)", but not "extern int foo(x);"
594         # XXX - this still trips over too many legitimate things,
595         # like "int foo(x,\n\ty);"
596 #               if (/^(\w+(\s|\*)+)+\w+\(/ && !/\)[;,](\s|^A)*$/ &&
597 #                   !/^(extern|static)\b/) {
598 #                       err("return type of function not on separate line");
599 #               }
600         # this is a close approximation
601         if (/^(\w+(\s|\*)+)+\w+\(.*\)(\s|^A)*$/ &&
602             !/^(extern|static)\b/) {
603                 err("return type of function not on separate line");
604         }
605         if (/^#define /) {
606                 err("#define followed by space instead of tab");
607         }
608         if (/^\s*return\W[^;]*;/ && !/^\s*return\s*\(.*\);/) {
609                 err("unparenthesized return expression");
610         }
611         if (/\bsizeof\b/ && !/\bsizeof\s*\(.*\)/) {
612                 err("unparenthesized sizeof expression");
613         }
614         if (/\(\s/) {
615                 err("whitespace after left paren");
616         }
617         # allow "for" statements to have empty "continue" clauses
618         if (/\s\)/ && !/^\s*for \([^;]*;[^;]*; \)/) {
619                 err("whitespace before right paren");
620         }
621         if (/^\s*\(void\)[^ ]/) {
622                 err("missing space after (void) cast");
623         }
624         if (/\S{/ && !/{{/) {
625                 err("missing space before left brace");
626         }
627         if ($in_function && /^\s+{/ &&
628             ($prev =~ /\)\s*$/ || $prev =~ /\bstruct\s+\w+$/)) {
629                 err("left brace starting a line");
630         }
631         if (/}(else|while)/) {
632                 err("missing space after right brace");
633         }
634         if (/}\s\s+(else|while)/) {
635                 err("extra space after right brace");
636         }
637         if (/\b_VOID\b|\bVOID\b|\bSTATIC\b/) {
638                 err("obsolete use of VOID or STATIC");
639         }
640         if (/\b$typename\*/o) {
641                 err("missing space between type name and *");
642         }
643         if (/^\s+#/) {
644                 err("preprocessor statement not in column 1");
645         }
646         if (/^#\s/) {
647                 err("blank after preprocessor #");
648         }
649         if (/!\s*(strcmp|strncmp|bcmp)\s*\(/) {
650                 err("don't use boolean ! with comparison functions");
651         }

653         #
654         # We completely ignore, for purposes of indentation:
655         #  * lines outside of functions
656         #  * preprocessor lines
```

```
657         #
658         if ($check_continuation && $in_function && !$in_cpp) {
659                 process_indent($_);
660         }
661         if ($picky) {
662                 # try to detect spaces after casts, but allow (e.g.)
663                 # "sizeof (int) + 1", "void (*funcptr)(int) = foo;", and
664                 # "int foo(int) __NORETURN;"
665                 if ((/^\($typename( \*+)?\)\s/o ||
666                     /\W\($typename( \*+)?\)\s/o) &&
667                     !/sizeof\s*\($typename( \*)?\)\s/o &&
668                     !/\($typename( \*+)?\)\s+=[^=]/o) {
669                         err("space after cast");
670                 }
671                 if (/\b$typename\s*\*\s/o &&
672                     !/\b$typename\s*\*\s+const\b/o) {
673                         err("unary * followed by space");
674                 }
675         }
676         if ($check_posix_types) {
677                 # try to detect old non-POSIX types.
678                 # POSIX requires all non-standard typedefs to end in _t,
679                 # but historically these have been used.
680                 if (/\b(unchar|ushort|uint|ulong|u_int|u_short|u_long|u_char|qua
681                         err("non-POSIX typedef $1 used: use $old2posix{$1} inste
682                 }
683         }
684         if ($heuristic) {
685                 # cannot check this everywhere due to "struct {\n...\n} foo;"
686                 if ($in_function && !$in_declaration &&
687                     /}./ && !/}\s+=/ && !/{.*}[;,]$/ && !/}(\s|^A)*$/ &&
688                     !/} (else|while)/ && !/}}/) {
689                         err("possible bad text following right brace");
690                 }
691                 # cannot check this because sub-blocks in
692                 # the middle of code are ok
693                 if ($in_function && /^\s+{/) {
694                         err("possible left brace starting a line");
695                 }
696         }
697         if (/^\s*else\W/) {
698                 if ($prev =~ /^\s*}$/) {
699                         err_prefix($prev,
700                             "else and right brace should be on same line");
701                 }
702         }
703         $prev = $line;
704 }

706 if ($prev eq "") {
707         err("last line in file is blank");
708 }

710 }

712 #
713 # Continuation-line checking
714 #
715 # The rest of this file contains the code for the continuation checking
716 # engine.  It's a pretty simple state machine which tracks the expression
717 # depth (unmatched '('s and '['s).
718 #
719 # Keep in mind that the argument to process_indent() has already been heavily
720 # processed; all comments have been replaced by control-A, and the contents of
721 # strings and character constants have been elided.
722 #
```



```
724 my $cont_in;            # currently inside of a continuation
725 my $cont_off;           # skipping an initializer or definition
726 my $cont_noerr;         # suppress cascading errors
727 my $cont_start;         # the line being continued
728 my $cont_base;          # the base indentation
729 my $cont_first;         # this is the first line of a statement
730 my $cont_multiseg;      # this continuation has multiple segments

732 my $cont_special;       # this is a C statement (if, for, etc.)
733 my $cont_macro;         # this is a macro
734 my $cont_case;          # this is a multi-line case

736 my @cont_paren;         # the stack of unmatched ( and [s we've seen

738 sub
739 reset_indent()
740 {
741         $cont_in = 0;
742         $cont_off = 0;
743 }

745 sub
746 delabel($)
747 {
748         #
749         # replace labels with tabs.  Note that there may be multiple
750         # labels on a line.
751         #
752         local $_ = $_[0];

754         while (/^(\t*)( *(?:(?:\w+\s*)|(?:case\b[^:]*)): *)(.*)$/) {
755                 my ($pre_tabs, $label, $rest) = ($1, $2, $3);
756                 $_ = $pre_tabs;
757                 while ($label =~ s/^([^\t]*)(\t+)//) {
758                         $_ .= "\t" x (length($2) + length($1) / 8);
759                 }
760                 $_ .= ("\t" x (length($label) / 8)).$rest;
761         }

763         return ($_);
764 }

766 sub
767 process_indent($)
768 {
769         require strict;
770         local $_ = $_[0];                       # preserve the global $_

772         s/^A//g; # No comments
773         s/\s+$//;       # Strip trailing whitespace

775         return                  if (/^$/);      # skip empty lines

777         # regexps used below; keywords taking (), macros, and continued cases
778         my $special = '(?:(?:\}\s*)?else\s+)?(?:if|for|while|switch)\b';
779         my $macro = '[A-Z_][A-Z_0-9]*\(';
780         my $case = 'case\b[^:]*$';

782         # skip over enumerations, array definitions, initializers, etc.
783         if ($cont_off <= 0 && !/^\s*$special/ &&
784             (/(?:(?:\b(?:enum|struct|union)\s*[^\{]*)|(?:\s+=\s*)){/ ||
785              (/^\s*{/ && $prev =~ /=\s*(?:\/\*.*\*\/\s*)*$/))) {
786                 $cont_in = 0;
787                 $cont_off = tr/{/{/ - tr/}/}/;
788                 return;
```

```
789         }
790         if ($cont_off) {
791                 $cont_off += tr/{/{/ - tr/}/}/;
792                 return;
793         }

795         if (!$cont_in) {
796                 $cont_start = $line;

798                 if (/^\t* /) {
799                         err("non-continuation indented 4 spaces");
800                         $cont_noerr = 1;                # stop reporting
801                 }
802                 $_ = delabel($_);       # replace labels with tabs

804                 # check if the statement is complete
805                 return          if (/^\s*\}?$/);
806                 return          if (/^\s*\}?\s*else\s*\{?$/);
807                 return          if (/^\s*do\s*\{?$/);
808                 return          if (/{$/);
809                 return          if (/}[,;]?$/);

811                 # Allow macros on their own lines
812                 return          if (/^\s*[A-Z_][A-Z_0-9]*$/);

814                 # cases we don't deal with, generally non-kosher
815                 if (/{/) {
816                         err("stuff after {");
817                         return;
818                 }

820                 # Get the base line, and set up the state machine
821                 /^(\t*)/;
822                 $cont_base = $1;
823                 $cont_in = 1;
824                 @cont_paren = ();
825                 $cont_first = 1;
826                 $cont_multiseg = 0;

828                 # certain things need special processing
829                 $cont_special = /^\s*$special/? 1 : 0;
830                 $cont_macro = /^\s*$macro/? 1 : 0;
831                 $cont_case = /^\s*$case/? 1 : 0;
832         } else {
833                 $cont_first = 0;

835                 # Strings may be pulled back to an earlier (half-)tabstop
836                 unless ($cont_noerr || /^$cont_base    / ||
837                     (/^\t*(?:    )?(?:gettext\()?\"/ && !/^$cont_base\t/)) {
838                         err_prefix($cont_start,
839                             "continuation should be indented 4 spaces");
840                 }
841         }

843         my $rest = $_;                  # keeps the remainder of the line

845         #
846         # The split matches 0 characters, so that each 'special' character
847         # is processed separately.  Parens and brackets are pushed and
848         # popped off the @cont_paren stack.  For normal processing, we wait
849         # until a ; or { terminates the statement.  "special" processing
850         # (if/for/while/switch) is allowed to stop when the stack empties,
851         # as is macro processing.  Case statements are terminated with a :
852         # and an empty paren stack.
853         #
854         foreach $_ (split /[^\(\)\[\]\{\}\;\:]*/) {
```



```
855                 next            if (length($_) == 0);

857                 # rest contains the remainder of the line
858                 my $rxp = "[^\Q$_\E]*\Q$_\E";
859                 $rest =~ s/^$rxp//;

861                 if (/\(/ || /\[/) {
862                         push @cont_paren, $_;
863                 } elsif (/\)/ || /\]/) {
864                         my $cur = $_;
865                         tr/\)\]/\(\[/;

867                         my $old = (pop @cont_paren);
868                         if (!defined($old)) {
869                                 err("unexpected '$cur'");
870                                 $cont_in = 0;
871                                 last;
872                         } elsif ($old ne $_) {
873                                 err("'$cur' mismatched with '$old'");
874                                 $cont_in = 0;
875                                 last;
876                         }

878                         #
879                         # If the stack is now empty, do special processing
880                         # for if/for/while/switch and macro statements.
881                         #
882                         next            if (@cont_paren != 0);
883                         if ($cont_special) {
884                                 if ($rest =~ /^\s*{?$/) {
885                                         $cont_in = 0;
886                                         last;
887                                 }
888                                 if ($rest =~ /^\s*;$/) {
889                                         err("empty if/for/while body ".
890                                             "not on its own line");
891                                         $cont_in = 0;
892                                         last;
893                                 }
894                                 if (!$cont_first && $cont_multiseg == 1) {
895                                         err_prefix($cont_start,
896                                             "multiple statements continued ".
897                                             "over multiple lines");
898                                         $cont_multiseg = 2;
899                                 } elsif ($cont_multiseg == 0) {
900                                         $cont_multiseg = 1;
901                                 }
902                                 # We've finished this section, start
903                                 # processing the next.
904                                 goto section_ended;
905                         }
906                         if ($cont_macro) {
907                                 if ($rest =~ /^$/) {
908                                         $cont_in = 0;
909                                         last;
910                                 }
911                         }
912                 } elsif (/\;/) {
913                         if ($cont_case) {
914                                 err("unexpected ;");
915                         } elsif (!$cont_special) {
916                                 err("unexpected ;")     if (@cont_paren != 0);
917                                 if (!$cont_first && $cont_multiseg == 1) {
918                                         err_prefix($cont_start,
919                                             "multiple statements continued ".
920                                             "over multiple lines");
```

```
921                                     $cont_multiseg = 2;
922                             } elsif ($cont_multiseg == 0) {
923                                     $cont_multiseg = 1;
924                             }
925                             if ($rest =~ /^$/) {
926                                     $cont_in = 0;
927                                     last;
928                             }
929                             if ($rest =~ /^\s*special/) {
930                                     err("if/for/while/switch not started ".
931                                         "on its own line");
932                             }
933                             goto section_ended;
934                     }
935                 } elsif (/\{/) {
936                         err("{ while in parens/brackets") if (@cont_paren != 0);
937                         err("stuff after {")            if ($rest =~ /[^\s}]/);
938                         $cont_in = 0;
939                         last;
940                 } elsif (/\}/) {
941                         err("} while in parens/brackets") if (@cont_paren != 0);
942                         if (!$cont_special && $rest !~ /^\s*(while|else)\b/) {
943                                 if ($rest =~ /^$/) {
944                                         err("unexpected }");
945                                 } else {
946                                         err("stuff after }");
947                                 }
948                                 $cont_in = 0;
949                                 last;
950                         }
951                 } elsif (/\:/ && $cont_case && @cont_paren == 0) {
952                         err("stuff after multi-line case") if ($rest !~ /$^/);
953                         $cont_in = 0;
954                         last;
955                 }
956                 next;
957 section_ended:
958                 # End of a statement or if/while/for loop.  Reset
959                 # cont_special and cont_macro based on the rest of the
960                 # line.
961                 $cont_special = ($rest =~ /^\s*$special/)? 1 : 0;
962                 $cont_macro = ($rest =~ /^\s*$macro/)? 1 : 0;
963                 $cont_case = 0;
964                 next;
965         }
966         $cont_noerr = 0                 if (!$cont_in);
967 }
```