```
**********************************************************
   24998 Mon Nov 24 19:05:56 2014
new/usr/src/tools/scripts/cstyle.pl
patch cstyle-atomic
**********************************************************
```

```
#!/usr/bin/perl -w
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# @(#)cstyle 1.58 98/09/09 (from shannon)
#ident  "%Z%%M% %I%     %E% SMI"
#
# cstyle - check for some common stylistic errors.
#
#       cstyle is a sort of "lint" for C coding style.
#       It attempts to check for the style used in the
#       kernel, sometimes known as "Bill Joy Normal Form".
#
#       There's a lot this can't check for, like proper indentation
#       of code blocks.  There's also a lot more this could check for.
#
#       A note to the non perl literate:
#
#               perl regular expressions are pretty much like egrep
#               regular expressions, with the following special symbols
#
#               \s      any space character
#               \S      any non-space character
#               \w      any "word" character [a-zA-Z0-9_]
#               \W      any non-word character
#               \d      a digit [0-9]
#               \D      a non-digit
#               \b      word boundary (between \w and \W)
#               \B      non-word boundary
#

require 5.0;
use IO::File;
use Getopt::Std;
use strict;

my $usage =
"usage: cstyle [-chpvCP] [-o constructs] file ...
        -c      check continuation indentation inside functions
        -h      perform heuristic checks that are sometimes wrong
```



```
        -p      perform some of the more picky checks
        -v      verbose
        -C      don't check anything in header block comments
        -P      check for use of non-POSIX types
        -o constructs
                allow a comma-seperated list of optional constructs:
                    doxygen     allow doxygen-style block comments (/** /*!)
                    splint      allow splint-style lint comments (/*@ ... @*/)
";

my %opts;

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

sub cstyle($$) {

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

my $in_cpp = 0;
my $next_in_cpp = 0;

my $in_comment = 0;
my $in_header_comment = 0;
my $comment_done = 0;
my $in_warlock_comment = 0;
my $in_function = 0;
my $in_function_header = 0;
my $in_declaration = 0;
my $note_level = 0;
my $nextok = 0;
my $nocheck = 0;

my $in_string = 0;

my ($okmsg, $comment_prefix);

$line = '';
$prev = '';
reset_indent();

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

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

        $line = $_;

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

        #
        # normal strings and characters
        #
```

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



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

```
394         # a blank line terminates the declarations within a function.
395         # XXX - but still a problem in sub-blocks.
396         if ($in_declaration && /^$/) {
397                 $in_declaration = 0;
398         }

400         if ($comment_done) {
401                 $in_comment = 0;
402                 $in_header_comment = 0;
403                 $comment_done = 0;
404         }
405         # does this looks like the start of a block comment?
406         if (/$hdr_comment_start/) {
407                 if (!/^\t*\/\*/) {
408                         err("block comment not indented by tabs");
409                 }
410                 $in_comment = 1;
411                 /^(\s*)\//;
412                 $comment_prefix = $1;
413                 if ($comment_prefix eq "") {
414                         $in_header_comment = 1;
415                 }
416                 $prev = $line;
417                 next line;
418         }
419         # are we still in the block comment?
420         if ($in_comment) {
421                 if (/^$comment_prefix \*\/$/) {
422                         $comment_done = 1;
423                 } elsif (/\*\//) {
424                         $comment_done = 1;
425                         err("improper block comment close")
426                             unless ($ignore_hdr_comment && $in_header_comment);
427                 } elsif (!/^$comment_prefix \*[ \t]/ &&
428                     !/^$comment_prefix \*$/) {
429                         err("improper block comment")
430                             unless ($ignore_hdr_comment && $in_header_comment);
431                 }
432         }

434         if ($in_header_comment && $ignore_hdr_comment) {
435                 $prev = $line;
436                 next line;
437         }

439         # check for errors that might occur in comments and in code.

441         # allow spaces to be used to draw pictures in header comments.
442         if (/[^ ]     / && !/".*     .*"/ && !$in_header_comment) {
443                 err("spaces instead of tabs");
444         }
445         if (/^ / && !/^ \*[ \t\/]/ && !/^ \*$/ &&
446             (!/^    \w/ || $in_function != 0)) {
447                 err("indent by spaces instead of tabs");
448         }
449         if (/^\t+ [^ \t\*]/ || /^\t+  \S/ || /^\t+   \S/) {
450                 err("continuation line not indented by 4 spaces");
451         }
452         if (/$warlock_re/ && !/\*\//) {
453                 $in_warlock_comment = 1;
454                 $prev = $line;
455                 next line;
456         }
457         if (/^\s*\/\*./ && !/^\s*\/\*.*\*\// && !/$hdr_comment_start/) {
458                 err("improper first line of block comment");
```



```
459         }

461         if ($in_comment) {      # still in comment, don't do further checks
462                 $prev = $line;
463                 next line;
464         }

466         if ((/[^(]\/\*\S/ || /^\/\*\S/) &&
467             !(/$lint_re/ || ($splint_comments && /$splint_re/))) {
468                 err("missing blank after open comment");
469         }
470         if (/\S\*\/[^)]|\S\*\/$/ &&
471             !(/$lint_re/ || ($splint_comments && /$splint_re/))) {
472                 err("missing blank before close comment");
473         }
474         if (/\/\/\S/) {         # C++ comments
475                 err("missing blank after start comment");
476         }
477         # check for unterminated single line comments, but allow them when
478         # they are used to comment out the argument list of a function
479         # declaration.
480         if (/\S.*\/\*/ && !/\S.*\/\*.*\*\// && !/\(\/\*/) {
481                 err("unterminated single line comment");
482         }

484         if (/^(#else|#endif|#include)(.*)$/) {
485                 $prev = $line;
486                 if ($picky) {
487                         my $directive = $1;
488                         my $clause = $2;
489                         # Enforce ANSI rules for #else and #endif: no noncomment
490                         # identifiers are allowed after #endif or #else.  Allow
491                         # C++ comments since they seem to be a fact of life.
492                         if ((($1 eq "#endif") || ($1 eq "#else")) &&
493                             ($clause ne "") &&
494                             (!($clause =~ /^\s+\/\*.*\*\/$/)) &&
495                             (!($clause =~ /^\s+\/\/.*$/))) {
496                                 err("non-comment text following " .
497                                     "$directive (or malformed $directive " .
498                                     "directive)");
499                         }
500                 }
501                 next line;
502         }

504         #
505         # delete any comments and check everything else.  Note that
506         # ".*?" is a non-greedy match, so that we don't get confused by
507         # multiple comments on the same line.
508         #
509         s/\/\*.*?\*\//^A/g;
510         s/\/\/.*$/^A/;          # C++ comments

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

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

517         if (/[^<>\s][!<>=]=/ || /[^<>][!<>=]=[^\s,]/ ||
518             (/[^->]>[^,=>\s]/ && !/[^->]>$/) ||
519             (/[^<]<[^,=<\s]/ && !/[^<]<$/) ||
520             /[^<\s]<[^<]/ || /[^->\s]>[^>]/) {
521                 err("missing space around relational operator");
522         }
523         if (/\S>>=/ || /\S<<=/ || />>=\S/ || /<<=\S/ || /\S[-+*\/&|^%]=/ ||
524             (/[^-+*\/&|^%!<>=\s]=[^=]/ && !/[^-+*\/&|^%!<>=\s]=$/) ||
```

```
525             (/[^!<>=]=[^=\s]/ && !/[^!<>=]=$/)) {
526                 # XXX - should only check this for C++ code
527                 # XXX - there are probably other forms that should be allowed
528                 if (!/\soperator=/) {
529                         err("missing space around assignment operator");
530                 }
531         }
532         if (/[,;]\S/ && !/\bfor \(;;\)/) {
533                 err("comma or semicolon followed by non-blank");
534         }
535         # allow "for" statements to have empty "while" clauses
536         if (/\s[,;]/ && !/^[\t]+;$/ && !/^\s*for \([^;]*; ;[^;]*\)/) {
537                 err("comma or semicolon preceded by blank");
538         }
539         if (/^\s*(&&|\|\|)/) {
540                 err("improper boolean continuation");
541         }
542         if (/\S   *(&&|\|\|)/ || /(&&|\|\|)   *\S/) {
543                 err("more than one space around boolean operator");
544         }
545         if (/\b(for|if|while|switch|sizeof|return|case)\(/) {
546                 err("missing space between keyword and paren");
547         }
548         if (/(\b(for|if|while|switch|return)\b.*){2,}/ && !/^#define/) {
549                 # multiple "case" and "sizeof" allowed
550                 err("more than one keyword on line");
551         }
552         if (/\b(for|if|while|switch|sizeof|return|case)\s\s+\(/ &&
553             !/^#if\s+\(/) {
554                 err("extra space between keyword and paren");
555         }
556         # try to detect "func (x)" but not "if (x)" or
557         # "#define foo (x)" or "int (*func)();"
558         if (/\w\s\(/) {
559                 my $s = $_;
560                 # strip off all keywords on the line
561                 s/\b(for|if|while|switch|return|case|sizeof)\s\(/XXX(/g;
562                 s/#elif\s\(/XXX(/g;
563                 s/^#define\s+\w+\s+\(/XXX(/;
564                 # do not match things like "void (*f)();"
565                 # or "typedef void (func_t)();"
566                 s/\w\s\(+\*/XXX(*/g;
567                 s/\b($typename|void)\s+\(+/XXX(/og;
568                 if (/\w\s\(/) {
569                         err("extra space between function name and left paren");
570                 }
571                 $_ = $s;
572         }
573         # try to detect "int foo(x)", but not "extern int foo(x);"
574         # XXX - this still trips over too many legitimate things,
575         # like "int foo(x,\n\ty);"
576 #               if (/^(\w+(\s|\*)+)+\w+\(/ && !/\)[;,](\s|^A)*$/ &&
577 #                   !/^(extern|static)\b/) {
578 #                       err("return type of function not on separate line");
579 #               }
580         # this is a close approximation
581         if (/^(\w+(\s|\*)+)+\w+\(.*\)(\s|^A)*$/ &&
582             !/^(extern|static)\b/) {
583                 err("return type of function not on separate line");
584         }
585         if (/^#define /) {
586                 err("#define followed by space instead of tab");
587         }
588         if (/^\s*return\W[^;]*;/ && !/^\s*return\s*\(.*\);/) {
589                 err("unparenthesized return expression");
590         }
```



```
591         if (/\bsizeof\b/ && !/\bsizeof\s*\(.*\)/) {
592                 err("unparenthesized sizeof expression");
593         }
594         if (/\(\s/) {
595                 err("whitespace after left paren");
596         }
597         # allow "for" statements to have empty "continue" clauses
598         if (/\s\)/ && !/^\s*for \([^;]*;[^;]*; \)/) {
599                 err("whitespace before right paren");
600         }
601         if (/^\s*\(void\)[^ ]/) {
602                 err("missing space after (void) cast");
603         }
604         if (/\S{/ && !/{{/) {
605                 err("missing space before left brace");
606         }
607         if ($in_function && /^\s+{/ &&
608             ($prev =~ /\)\s*$/ || $prev =~ /\bstruct\s+\w+$/)) {
609                 err("left brace starting a line");
610         }
611         if (/}(else|while)/) {
612                 err("missing space after right brace");
613         }
614         if (/}\s\s+(else|while)/) {
615                 err("extra space after right brace");
616         }
617         if (/\b_VOID\b|\bVOID\b|\bSTATIC\b/) {
618                 err("obsolete use of VOID or STATIC");
619         }
620         if (/\b$typename\*/o) {
621                 err("missing space between type name and *");
622         }
623         if (/^\s+#/) {
624                 err("preprocessor statement not in column 1");
625         }
626         if (/^#\s/) {
627                 err("blank after preprocessor #");
628         }
629         if (/!\s*(strcmp|strncmp|bcmp)\s*\(/) {
630                 err("don't use boolean ! with comparison functions");
631         }
632         if (/\batomic_add_(8|16|32|64|char|short|int|long)\([^,]*,\s*1\)/) {
633                 err("use atomic_inc_*(...) instead of atomic_add_*(..., 1)");
634         }
635         if (/\batomic_add_(8|16|32|64|char|short|int|long)\([^,]*,\s*-1\)/) {
636                 err("use atomic_dec_*(...) instead of atomic_add_*(..., -1)");
637 #endif /* ! codereview */
638         }

640         #
641         # We completely ignore, for purposes of indentation:
642         #  * lines outside of functions
643         #  * preprocessor lines
644         #
645         if ($check_continuation && $in_function && !$in_cpp) {
646                 process_indent($_);
647         }
648         if ($picky) {
649                 # try to detect spaces after casts, but allow (e.g.)
650                 # "sizeof (int) + 1", "void (*funcptr)(int) = foo;", and
651                 # "int foo(int) __NORETURN;"
652                 if ((/^\($typename( \*+)?\)\s/o ||
653                     /\W\($typename( \*+)?\)\s/o) &&
654                     !/sizeof\s*\($typename( \*)?\)\s/o &&
655                     !/\($typename( \*+)?\)\s+=[^=]/o) {
656                         err("space after cast");
```

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

693 if ($prev eq "") {
694         err("last line in file is blank");
695 }

697 }

699 #
700 # Continuation-line checking
701 #
702 # The rest of this file contains the code for the continuation checking
703 # engine.  It's a pretty simple state machine which tracks the expression
704 # depth (unmatched '('s and '['s).
705 #
706 # Keep in mind that the argument to process_indent() has already been heavily
707 # processed; all comments have been replaced by control-A, and the contents of
708 # strings and character constants have been elided.
709 #

711 my $cont_in;            # currently inside of a continuation
712 my $cont_off;           # skipping an initializer or definition
713 my $cont_noerr;         # suppress cascading errors
714 my $cont_start;         # the line being continued
715 my $cont_base;          # the base indentation
716 my $cont_first;         # this is the first line of a statement
717 my $cont_multiseg;      # this continuation has multiple segments

719 my $cont_special;       # this is a C statement (if, for, etc.)
720 my $cont_macro;         # this is a macro
721 my $cont_case;          # this is a multi-line case
```



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

725 sub
726 reset_indent()
727 {
728         $cont_in = 0;
729         $cont_off = 0;
730 }

732 sub
733 delabel($)
734 {
735         #
736         # replace labels with tabs.  Note that there may be multiple
737         # labels on a line.
738         #
739         local $_ = $_[0];

741         while (/^(\t*)( *(?:(?:\w+\s*)|(?:case\b[^:]*)): *)(.*)$/) {
742                 my ($pre_tabs, $label, $rest) = ($1, $2, $3);
743                 $_ = $pre_tabs;
744                 while ($label =~ s/^([^\t]*)(\t+)//) {
745                         $_ .= "\t" x (length($2) + length($1) / 8);
746                 }
747                 $_ .= ("\t" x (length($label) / 8)).$rest;
748         }

750         return ($_);
751 }

753 sub
754 process_indent($)
755 {
756         require strict;
757         local $_ = $_[0];                       # preserve the global $_

759         s/^A//g; # No comments
760         s/\s+$//;       # Strip trailing whitespace

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

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

769         # skip over enumerations, array definitions, initializers, etc.
770         if ($cont_off <= 0 && !/^\s*$special/ &&
771             (/(?:(?:\b(?:enum|struct|union)\s*[^\{]*)|(?:\s+=\s*)){/ ||
772             (/^\s*{/ && $prev =~ /=\s*(?:\/\*.*\*\/\s*)*$/))) {
773                 $cont_in = 0;
774                 $cont_off = tr/{/{/ - tr/}/}/;
775                 return;
776         }
777         if ($cont_off) {
778                 $cont_off += tr/{/{/ - tr/}/}/;
779                 return;
780         }

782         if (!$cont_in) {
783                 $cont_start = $line;

785                 if (/^\t* /) {
786                         err("non-continuation indented 4 spaces");
787                         $cont_noerr = 1;               # stop reporting
788                 }
```

```
789                 $_ = delabel($_);       # replace labels with tabs

791                 # check if the statement is complete
792                 return          if (/^\s*\}?$/);
793                 return          if (/^\s*\}?\s*else\s*\{?$/);
794                 return          if (/^\s*do\s*\{?$/);
795                 return          if (/{$/);
796                 return          if (/}[,;]?$/);

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

801                 # cases we don't deal with, generally non-kosher
802                 if (/{/) {
803                         err("stuff after {");
804                         return;
805                 }

807                 # Get the base line, and set up the state machine
808                 /^(\t*)/;
809                 $cont_base = $1;
810                 $cont_in = 1;
811                 @cont_paren = ();
812                 $cont_first = 1;
813                 $cont_multiseg = 0;

815                 # certain things need special processing
816                 $cont_special = /^\s*$special/? 1 : 0;
817                 $cont_macro = /^\s*$macro/? 1 : 0;
818                 $cont_case = /^\s*$case/? 1 : 0;
819         } else {
820                 $cont_first = 0;

822                 # Strings may be pulled back to an earlier (half-)tabstop
823                 unless ($cont_noerr || /^$cont_base    / ||
824                     (/^\t*(?:    )?(?:gettext\()?\"/ && !/^$cont_base\t/)) {
825                         err_prefix($cont_start,
826                             "continuation should be indented 4 spaces");
827                 }
828         }

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

832         #
833         # The split matches 0 characters, so that each 'special' character
834         # is processed separately.  Parens and brackets are pushed and
835         # popped off the @cont_paren stack.  For normal processing, we wait
836         # until a ; or { terminates the statement.  "special" processing
837         # (if/for/while/switch) is allowed to stop when the stack empties,
838         # as is macro processing.  Case statements are terminated with a :
839         # and an empty paren stack.
840         #
841         foreach $_ (split /[^\(\)\[\]\{\}\;\:]*/) {
842                 next            if (length($_) == 0);

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

848                 if (/\(/ || /\[/) {
849                         push @cont_paren, $_;
850                 } elsif (/\)/ || /\]/) {
851                         my $cur = $_;
852                         tr/\)\]/\(\[/;

854                         my $old = (pop @cont_paren);
```



```
855                         if (!defined($old)) {
856                                 err("unexpected '$cur'");
857                                 $cont_in = 0;
858                                 last;
859                         } elsif ($old ne $_) {
860                                 err("'$cur' mismatched with '$old'");
861                                 $cont_in = 0;
862                                 last;
863                         }

865                         #
866                         # If the stack is now empty, do special processing
867                         # for if/for/while/switch and macro statements.
868                         #
869                         next            if (@cont_paren != 0);
870                         if ($cont_special) {
871                                 if ($rest =~ /^\s*{?$/) {
872                                         $cont_in = 0;
873                                         last;
874                                 }
875                                 if ($rest =~ /^\s*;$/) {
876                                         err("empty if/for/while body ".
877                                             "not on its own line");
878                                         $cont_in = 0;
879                                         last;
880                                 }
881                                 if (!$cont_first && $cont_multiseg == 1) {
882                                         err_prefix($cont_start,
883                                             "multiple statements continued ".
884                                             "over multiple lines");
885                                         $cont_multiseg = 2;
886                                 } elsif ($cont_multiseg == 0) {
887                                         $cont_multiseg = 1;
888                                 }
889                                 # We've finished this section, start
890                                 # processing the next.
891                                 goto section_ended;
892                         }
893                         if ($cont_macro) {
894                                 if ($rest =~ /^$/) {
895                                         $cont_in = 0;
896                                         last;
897                                 }
898                         }
899                 } elsif (/\;/) {
900                         if ($cont_case) {
901                                 err("unexpected ;");
902                         } elsif (!$cont_special) {
903                                 err("unexpected ;")     if (@cont_paren != 0);
904                                 if (!$cont_first && $cont_multiseg == 1) {
905                                         err_prefix($cont_start,
906                                             "multiple statements continued ".
907                                             "over multiple lines");
908                                         $cont_multiseg = 2;
909                                 } elsif ($cont_multiseg == 0) {
910                                         $cont_multiseg = 1;
911                                 }
912                                 if ($rest =~ /^$/) {
913                                         $cont_in = 0;
914                                         last;
915                                 }
916                                 if ($rest =~ /^\s*special/) {
917                                         err("if/for/while/switch not started ".
918                                             "on its own line");
919                                 }
920                                 goto section_ended;
```

```
921                         }
922                 } elsif (/\{/) {
923                         err("{ while in parens/brackets") if (@cont_paren != 0);
924                         err("stuff after {")            if ($rest =~ /[^\s}]/);
925                         $cont_in = 0;
926                         last;
927                 } elsif (/\}/) {
928                         err("} while in parens/brackets") if (@cont_paren != 0);
929                         if (!$cont_special && $rest !~ /^\s*(while|else)\b/) {
930                                 if ($rest =~ /^$/) {
931                                         err("unexpected }");
932                                 } else {
933                                         err("stuff after }");
934                                 }
935                                 $cont_in = 0;
936                                 last;
937                         }
938                 } elsif (/\:/ && $cont_case && @cont_paren == 0) {
939                         err("stuff after multi-line case") if ($rest !~ /$^/);
940                         $cont_in = 0;
941                         last;
942                 }
943                 next;
944 section_ended:
945                 # End of a statement or if/while/for loop.  Reset
946                 # cont_special and cont_macro based on the rest of the
947                 # line.
948                 $cont_special = ($rest =~ /^\s*$special/)? 1 : 0;
949                 $cont_macro = ($rest =~ /^\s*$macro/)? 1 : 0;
950                 $cont_case = 0;
951                 next;
952         }
953         $cont_noerr = 0                 if (!$cont_in);
954 }
```