****
25217 Mon Jan 5 11:35:28 2015
new/usr/src/tools/scripts/cstyle.pl
patch cstyle-x86
****

```
#!/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) = @_;
$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
        #
        s/’([^\\’]|\\[^xX0]|\\0[0-9]*|\\[xX][0-9a-fA-F]*)’/’’/g;
        s/"([^\\"]|\\.)*"/\"\"/g;

        #
        # detect string continuation
        #
        if ($nocheck || $in_comment) {
                $in_string = 0;
        } else {
                #
                # Now that all full strings are replaced with "", we check
                # for unfinished strings continuing onto the next line.
                #
                $in_string =
                    (s/([^"](?:"")*)"([^\\"]|\\.)*\\$/$1""/ ||
                    s/^("")*"([^\\"]|\\.)*\\$/""/);
        }

        #
        # figure out if we are in a cpp directive
        #
        $in_cpp = $next_in_cpp || /^\s*#/;      # continued or started
        $next_in_cpp = $in_cpp && /\\$/;        # only if continued

        # strip off trailing backslashes, which appear in long macros
        s/\s*\\$//;

        # an /* END CSTYLED */ comment ends a no-check block.
        if ($nocheck) {
                if (/\/\* *END *CSTYLED *\*\//) {
                        $nocheck = 0;
                } else {
                        reset_indent();
                        next line;
                }
        }

        # a /*CSTYLED*/ comment indicates that the next line is ok.
        if ($nextok) {
                if ($okmsg) {
                        err($okmsg);
                }
                $nextok = 0;
                $okmsg = 0;
                if (/\/\* *CSTYLED.*\*\//) {
                        /^.*\/\* *CSTYLED *(.*) *\*\/.*$/;
                        $okmsg = $1;
                        $nextok = 1;
                }
                $no_errs = 1;
        } elsif ($no_errs) {
                $no_errs = 0;
        }

        # check length of line.
        # first, a quick check to see if there is any chance of being too long.
        if (($line =~ tr/\t/\t/) * 7 + length($line) > 80) {
                # yes, there is a chance.
                # replace tabs with spaces and check again.
                my $eline = $line;
                1 while $eline =~
                    s/\t+/’ ’ x (length($&) * 8 - length($‘) % 8)/e;
                if (length($eline) > 80) {
                        err("line > 80 characters");
                }
        }
```

```
        # ignore NOTE(...) annotations (assumes NOTE is on lines by itself).
        if ($note_level || /\b_?NOTE\s*\(/) { # if in NOTE or this is NOTE
                s/[^()]//g;                       # eliminate all non-parens
                $note_level += s/\(//g - length;  # update paren nest level
                next;
        }

        # a /* BEGIN CSTYLED */ comment starts a no-check block.
        if (/\/\* *BEGIN *CSTYLED *\*\//) {
                $nocheck = 1;
        }

        # a /*CSTYLED*/ comment indicates that the next line is ok.
        if (/\/\* *CSTYLED.*\*\//) {
                /^.*\/\* *CSTYLED *(.*) *\*\/.*$/;
                $okmsg = $1;
                $nextok = 1;
        }
        if (/\/\/ *CSTYLED/) {
                /^.*\/\/ *CSTYLED *(.*)$/;
                $okmsg = $1;
                $nextok = 1;
        }

        # universal checks; apply to everything
        if (/\t +\t/) {
                err("spaces between tabs");
        }
        if (/ \t+ /) {
                err("tabs between spaces");
        }
        if (/\s$/) {
                err("space or tab at end of line");
        }
        if (/[^ \t(]\/\*/ && !/\w\(\/\*.*\*\/\);/) {
                err("comment preceded by non-blank");
        }

        # is this the beginning or ending of a function?
        # (not if "struct foo\n{\n")
        if (/^{$/ && $prev =~ /\)\s*(const\s*)?(\/\*.*\*\/\s*)?\\?$/) {
                $in_function = 1;
                $in_declaration = 1;
                $in_function_header = 0;
                $prev = $line;
                next line;
        }
        if (/^}\s*(\/\*.*\*\/\s*)*$/) {
                if ($prev =~ /^\s*return\s*;/) {
                        err_prev("unneeded return at end of function");
                }
                $in_function = 0;
                reset_indent();         # we don’t check between functions
                $prev = $line;
                next line;
        }
        if (/^\w*\($/) {
                $in_function_header = 1;
        }

        if ($in_warlock_comment && /\*\//) {
                $in_warlock_comment = 0;
                $prev = $line;
                next line;
        }
```

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

	if ($comment_done) {
		$in_comment = 0;
		$in_header_comment = 0;
		$comment_done = 0;
	}
	# does this looks like the start of a block comment?
	if (/$hdr_comment_start/) {
		if (!/^\t*\/\*/) {
			err("block comment not indented by tabs");
		}
		$in_comment = 1;
		/^(\s*)\//;
		$comment_prefix = $1;
		if ($comment_prefix eq "") {
			$in_header_comment = 1;
		}
		$prev = $line;
		next line;
	}
	# are we still in the block comment?
	if ($in_comment) {
		if (/^$comment_prefix \*\/$/) {
			$comment_done = 1;
		} elsif (/\*\//) {
			$comment_done = 1;
			err("improper block comment close")
			    unless ($ignore_hdr_comment && $in_header_comment);
		} elsif (!/^$comment_prefix \*[ \t]/ &&
		    !/^$comment_prefix \*$/) {
			err("improper block comment")
			    unless ($ignore_hdr_comment && $in_header_comment);
		}
	}

	if ($in_header_comment && $ignore_hdr_comment) {
		$prev = $line;
		next line;
	}

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

	# allow spaces to be used to draw pictures in header comments.
	if (/[^ ]     / && !/".*     .*"/ && !$in_header_comment) {
		err("spaces instead of tabs");
	}
	if (/^ / && !/^ \*[ \t\/]/ && !/^ \*$/ &&
	    (!/^    \w/ || $in_function != 0)) {
		err("indent by spaces instead of tabs");
	}
	if (/^\t+ [^ \t\*]/ || /^\t+  \S/ || /^\t+   \S/) {
		err("continuation line not indented by 4 spaces");
	}
	if (/$warlock_re/ && !/\*\//) {
		$in_warlock_comment = 1;
		$prev = $line;
		next line;
	}
	if (/^\s*\/\*./ && !/^\s*\/\*.*\*\// && !/$hdr_comment_start/) {
		err("improper first line of block comment");
```

```
	}

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

	if ((/[^(]\/\*\S/ || /^\/\*\S/) &&
	    !(/$lint_re/ || ($splint_comments && /$splint_re/))) {
		err("missing blank after open comment");
	}
	if (/\S\*\/[^)]|\S\*\/$/ &&
	    !(/$lint_re/ || ($splint_comments && /$splint_re/))) {
		err("missing blank before close comment");
	}
	if (/\/\/\S/) {		# C++ comments
		err("missing blank after start comment");
	}
	# check for unterminated single line comments, but allow them when
	# they are used to comment out the argument list of a function
	# declaration.
	if (/\S.*\/\*/ && !/\S.*\/\*.*\*\// && !/\(\/\*/) {
		err("unterminated single line comment");
	}

	# check that #if doesn't enumerate ISA defines when there are more
	# concise ways of checking.  E.g., don't do:
	#     #if defined(__amd64) || defined(__i386)
	# when there is:
	#     #ifdef __x86
	if (/^#if\sdefined\((.*)\)\s\|\|\sdefined\((.*)\)/) {
		my $first = $1;
		my $second = $2;
		($first, $second) = ($second, $first) if ($first gt $second);

		if (($first eq "__amd64") && ($second eq "__i386")) {
			err("#if checking for $first or $second instead of " .
			    "__x86");
		}
#endif /* ! codereview */
	}

	if (/^(#else|#endif|#include)(.*)$/) {
		$prev = $line;
		if ($picky) {
			my $directive = $1;
			my $clause = $2;
			# Enforce ANSI rules for #else and #endif: no noncomment
			# identifiers are allowed after #endif or #else.  Allow
			# C++ comments since they seem to be a fact of life.
			if ((($1 eq "#endif") || ($1 eq "#else")) &&
			    ($clause ne "") &&
			    (!($clause =~ /^\s+\/\*.*\*\/$/)) &&
			    (!($clause =~ /^\s+\/\/.*$/))) {
				err("non-comment text following " .
				    "$directive (or malformed $directive " .
				    "directive)");
			}
		}
		next line;
	}

	#
	# delete any comments and check everything else.  Note that
	# ".*?" is a non-greedy match, so that we don't get confused by
	# multiple comments on the same line.
```

```
        #
        s/\/\*.*?\*\//^A/g;
        s/\/\/.*$/^A/;                  # C++ comments

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

        # following checks do not apply to text in comments.

        if (/[^<>\s][!<>=]=/ || /[^<>][!<>=]=[^\s,]/ ||
            (/[^->]>[^,=>\s]/ && !/[^->]>$/) ||
            (/[^<]<[^,=<\s]/ && !/[^<]<$/) ||
            /[^<\s]<[^<]/ || /[^->\s]>[^>]/) {
                err("missing space around relational operator");
        }
        if (/\S>>=/ || /\S<<=/ || />>=\S/ || /<<=\S/ || /\S[-+*\/&|^%]=/ ||
            (/[^-+*\/&|^%!<>=\s]=[^=]/ && !/[^-+*\/&|^%!<>=\s]=$/) ||
            (/[^!<>=]=[^=\s]/ && !/[^!<>=]=$/)) {
                # XXX - should only check this for C++ code
                # XXX - there are probably other forms that should be allowed
                if (!/\soperator=/) {
                        err("missing space around assignment operator");
                }
        }
        if (/[,;]\S/ && !/\bfor \(;;\)/) {
                err("comma or semicolon followed by non-blank");
        }
        # allow "for" statements to have empty "while" clauses
        if (/\s[,;]/ && !/^[\t]+;$/ && !/^\s*for \([^;]*; ;[^;]*\)/) {
                err("comma or semicolon preceded by blank");
        }
        if (/^\s*(&&|\|\|)/) {
                err("improper boolean continuation");
        }
        if (/\S   *(&&|\|\|)/ || /(&&|\|\|)   *\S/) {
                err("more than one space around boolean operator");
        }
        if (/\b(for|if|while|switch|sizeof|return|case)\(/) {
                err("missing space between keyword and paren");
        }
        if (/(\b(for|if|while|switch|return)\b.*){2,}/ && !/^#define/) {
                # multiple "case" and "sizeof" allowed
                err("more than one keyword on line");
        }
        if (/\b(for|if|while|switch|sizeof|return|case)\s\s+\(/ &&
            !/^#if\s+\(/) {
                err("extra space between keyword and paren");
        }
        # try to detect "func (x)" but not "if (x)" or
        # "#define foo (x)" or "int (*func)();"
        if (/\w\s\(/) {
                my $s = $_;
                # strip off all keywords on the line
                s/\b(for|if|while|switch|return|case|sizeof)\s\(/XXX(/g;
                s/#elif\s\(/XXX(/g;
                s/^#define\s+\w+\s+\(/XXX(/;
                # do not match things like "void (*f)();"
                # or "typedef void (func_t)();"
                s/\w\s\(+\*/XXX(*/g;
                s/\b($typename|void)\s+\(+/XXX(/og;
                if (/\w\s\(/) {
                        err("extra space between function name and left paren");
                }
                $_ = $s;
        }
        # try to detect "int foo(x)", but not "extern int foo(x);"
```

```
        # XXX - this still trips over too many legitimate things,
        # like "int foo(x,\n\ty);"
#               if (/^(\w+(\s|\*)+)+\w+\(/ && !/\)[;,](\s|^A)*$/ &&
#                   !/^(extern|static)\b/) {
#                       err("return type of function not on separate line");
#               }
        # this is a close approximation
        if (/^(\w+(\s|\*)+)+\w+\(.*\)(\s|^A)*$/ &&
            !/^(extern|static)\b/) {
                err("return type of function not on separate line");
        }
        if (/^#define /) {
                err("#define followed by space instead of tab");
        }
        if (/^\s*return\W[^;]*;/ && !/^\s*return\s*\(.*\);/) {
                err("unparenthesized return expression");
        }
        if (/\bsizeof\b/ && !/\bsizeof\s*\(.*\)/) {
                err("unparenthesized sizeof expression");
        }
        if (/\(\s/) {
                err("whitespace after left paren");
        }
        # allow "for" statements to have empty "continue" clauses
        if (/\s\)/ && !/^\s*for \([^;]*;[^;]*; \)/) {
                err("whitespace before right paren");
        }
        if (/^\s*\(void\)[^ ]/) {
                err("missing space after (void) cast");
        }
        if (/\S{/ && !/{{/) {
                err("missing space before left brace");
        }
        if ($in_function && /^\s+{/ &&
            ($prev =~ /\)\s*$/ || $prev =~ /\bstruct\s+\w+$/)) {
                err("left brace starting a line");
        }
        if (/}(else|while)/) {
                err("missing space after right brace");
        }
        if (/}\s\s+(else|while)/) {
                err("extra space after right brace");
        }
        if (/\b_VOID\b|\bVOID\b|\bSTATIC\b/) {
                err("obsolete use of VOID or STATIC");
        }
        if (/\b$typename\*/o) {
                err("missing space between type name and *");
        }
        if (/^\s+#/) {
                err("preprocessor statement not in column 1");
        }
        if (/^#\s/) {
                err("blank after preprocessor #");
        }
        if (/!\s*(strcmp|strncmp|bcmp)\s*\(/) {
                err("don't use boolean ! with comparison functions");
        }

        #
        # We completely ignore, for purposes of indentation:
        #  * lines outside of functions
        #  * preprocessor lines
        #
        if ($check_continuation && $in_function && !$in_cpp) {
                process_indent($_);
        }
        if ($picky) {
                # try to detect spaces after casts, but allow (e.g.)
                # "sizeof (int) + 1", "void (*funcptr)(int) = foo;", and
                # "int foo(int) __NORETURN;"
                if ((/^\($typename( \*+)?\)\s/o ||
                    /\W\($typename( \*+)?\)\s/o) &&
                    !/sizeof\s*\($typename( \*)?\)\s/o &&
                    !/\($typename( \*+)?\)\s+=[^=]/o) {
                        err("space after cast");
                }
                if (/\b$typename\s*\*\s/o &&
                    !/\b$typename\s*\*\s+const\b/o) {
                        err("unary * followed by space");
                }
        }
        if ($check_posix_types) {
                # try to detect old non-POSIX types.
                # POSIX requires all non-standard typedefs to end in _t,
                # but historically these have been used.
                if (/\b(unchar|ushort|uint|ulong|u_int|u_short|u_long|u_char|qua
                        err("non-POSIX typedef $1 used: use $old2posix{$1} inste
                }
        }
        if ($heuristic) {
                # cannot check this everywhere due to "struct {\n...\n} foo;"
                if ($in_function && !$in_declaration &&
                    /}./ && !/}\s+=/ && !/{.*}[;,]$/ && !/}(\s|^A)*$/ &&
                    !/} (else|while)/ && !/}}/) {
                        err("possible bad text following right brace");
                }
                # cannot check this because sub-blocks in
                # the middle of code are ok
                if ($in_function && /^\s+{/) {
                        err("possible left brace starting a line");
                }
        }
        if (/^\s*else\W/) {
                if ($prev =~ /^\s*}$/) {
                        err_prefix($prev,
                            "else and right brace should be on same line");
                }
        }
        $prev = $line;
}

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

}

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

my $cont_in;            # currently inside of a continuation
my $cont_off;           # skipping an initializer or definition
```

```
my $cont_noerr;         # suppress cascading errors
my $cont_start;         # the line being continued
my $cont_base;          # the base indentation
my $cont_first;         # this is the first line of a statement
my $cont_multiseg;      # this continuation has multiple segments

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

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

sub
reset_indent()
{
        $cont_in = 0;
        $cont_off = 0;
}

sub
delabel($)
{
        #
        # replace labels with tabs.  Note that there may be multiple
        # labels on a line.
        #
        local $_ = $_[0];

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

        return ($_);
}

sub
process_indent($)
{
        require strict;
        local $_ = $_[0];                       # preserve the global $_

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

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

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

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

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

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

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

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

                # cases we don’t deal with, generally non-kosher
                if (/{/) {
                        err("stuff after {");
                        return;
                }

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

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

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

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

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

                # rest contains the remainder of the line
```

```
                my $rxp = "[^\Q$_\E]*\Q$_\E";
                $rest =~ s/^$rxp//;

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

                        my $old = (pop @cont_paren);
                        if (!defined($old)) {
                                err("unexpected ’$cur’");
                                $cont_in = 0;
                                last;
                        } elsif ($old ne $_) {
                                err("’$cur’ mismatched with ’$old’");
                                $cont_in = 0;
                                last;
                        }

                        #
                        # If the stack is now empty, do special processing
                        # for if/for/while/switch and macro statements.
                        #
                        next            if (@cont_paren != 0);
                        if ($cont_special) {
                                if ($rest =~ /^\s*{?$/) {
                                        $cont_in = 0;
                                        last;
                                }
                                if ($rest =~ /^\s*;$/) {
                                        err("empty if/for/while body ".
                                            "not on its own line");
                                        $cont_in = 0;
                                        last;
                                }
                                if (!$cont_first && $cont_multiseg == 1) {
                                        err_prefix($cont_start,
                                            "multiple statements continued ".
                                            "over multiple lines");
                                        $cont_multiseg = 2;
                                } elsif ($cont_multiseg == 0) {
                                        $cont_multiseg = 1;
                                }
                                # We’ve finished this section, start
                                # processing the next.
                                goto section_ended;
                        }
                        if ($cont_macro) {
                                if ($rest =~ /^$/) {
                                        $cont_in = 0;
                                        last;
                                }
                        }
                } elsif (/\;/) {
                        if ($cont_case) {
                                err("unexpected ;");
                        } elsif (!$cont_special) {
                                err("unexpected ;")     if (@cont_paren != 0);
                                if (!$cont_first && $cont_multiseg == 1) {
                                        err_prefix($cont_start,
                                            "multiple statements continued ".
                                            "over multiple lines");
                                        $cont_multiseg = 2;
                                } elsif ($cont_multiseg == 0) {
                                        $cont_multiseg = 1;
```

```
				}
				if ($rest =~ /^$/) {
					$cont_in = 0;
					last;
				}
				if ($rest =~ /^\s*special/) {
					err("if/for/while/switch not started ".
					    "on its own line");
				}
				goto section_ended;
			}
		} elsif (/\{/) {
			err("{ while in parens/brackets") if (@cont_paren != 0);
			err("stuff after {")		if ($rest =~ /[^\s}]/);
			$cont_in = 0;
			last;
		} elsif (/\}/) {
			err("} while in parens/brackets") if (@cont_paren != 0);
			if (!$cont_special && $rest !~ /^\s*(while|else)\b/) {
				if ($rest =~ /^$/) {
					err("unexpected }");
				} else {
					err("stuff after }");
				}
				$cont_in = 0;
				last;
			}
		} elsif (/\:/ && $cont_case && @cont_paren == 0) {
			err("stuff after multi-line case") if ($rest !~ /$^/);
			$cont_in = 0;
			last;
		}
		next;
section_ended:
		# End of a statement or if/while/for loop.  Reset
		# cont_special and cont_macro based on the rest of the
		# line.
		$cont_special = ($rest =~ /^\s*$special/)? 1 : 0;
		$cont_macro = ($rest =~ /^\s*$macro/)? 1 : 0;
		$cont_case = 0;
		next;
	}
	$cont_noerr = 0			if (!$cont_in);
}
```