CodeCheck

The Quick Reference Guide

By Abraxas Software, Inc.

Editor: Patrick Conley

Acknowledgments:
Thanks to the thousandss of CodeCheck customers who have given us feedback in the past fifteen years.

Abraxas Software, Inc.
4726 SE Division Street
Portland, Oregon 97206 USA

TEL (503) 232-0540 FAX (503) 232-0543

Email: support@abxsoft.com
WWW: www.abxsoft.com
FTP: www.abxsoft.com/dl
PDF: www.abxsoft.com/pdf

Printed in U.S.A.
Fifth Printing, September 2003

Table of Contents

Introduction - 4

Command Line Options - 5

File Name Conventions - 10

Variables and Functions - 11

CodeCheck Manifest Constants - 40

System Dependent Constants - 46

Glossary - 50

Introduction

CodeCheck is invoked by means of a command line with either of these formats:

```
        check -options foo.c
        check foo.c -options
```

In this command line format foo.c refers to the name of the C source file to be analyzed. Any number of source files may be specified, arbitrarily intermixed with options.

The rules that are to be used to perform this analysis can be specified in the options list, as described below. If no rule file is specified, CodeCheck will look for a precompiled rule file named default.cco, first in the current directory and then in the directories specified in the CCRULES environment variable. If this file is not found, CodeCheck will perform a simple syntactic scan of the source file without any user-defined rules.

To analyze a multiple-file project with CodeCheck, either list all of the source filenames on the command line, or create a new file containing the names of all of the source files (*excluding* the names of header files and libraries). Give this project file the extension .ccp. Then invoke CodeCheck, specifying the project file instead of a source file:

```
          check -options myproject.ccp
```

CodeCheck will apply its rules to each source file named in myproject.ccp, and will apply project-level checking across all the files in the project. The ccp extension informs CodeCheck that the specified file is a project file rather than a C source file. This extension may be omitted in the command-line. *Note:* the project file must end with a newline character. The file may contain switches and comments.

To specify a rule file. The name of the rule file must follow immediately, *e.g.* if the rule file name is foobar.cc and the C or C++ source filename is mysource.c: check -Rfoobar.cc mysource.c

Command Line Options

CodeCheck command-line options are not case-sensitive. The available options:

-B Instruct CodeCheck that braces are on the same nesting level as material surrounded by the braces. If this option is not specified, then CodeCheck assumes that the braces are at the previous nesting level. This option only affects the predefined variable lin_nest_level.

-C Suppress type checking.

-D Define a macro. The name of the macro must follow immediately. Thus

```
check -dDO_FOREVER=for(;;)
```

has the same effect as starting the source file with

```
#define DO_FOREVER for(;;)
```

Macros defined on the command-line may not have arguments.

-D? Show internal symbol table for all macros. (Debugging)

-E Do NOT ignore tokens that are derived from macro expansion when perform-
ing counts, e.g. of operators and operands. The default (-E not given)
is for CodeCheck to ignore all macro-derived tokens when counting.

-F Count tokens, lines, operators, or operands when reading header files.
The default (-F not specified) is for CodeCheck not to count tokens,
lines, operators, or operands when reading header files.

-G

-G Do not read each header file more than once per module. CAUTION: Some header files may be designed to be read multiple times, with conditional access to different sections of the header.

-I Specify a path to search when looking for header files. Use a separate -I for each path. The pathname must follow immediately, e.g.

```
check  -Iusr/metaware/headers  src.c
```

-H List lines from all header files in the listing file.

-J Suppress all error messages generated by CodeCheck. This option does not
affect warnings generated by CodeCheck rules.

-K Identify the dialect of C to be assumed for the source files. A digit
should follow immediately, corresponding to the dialect. The dialects of C/C++ that are currently supported include:

```
0  =>  K&R (1978) C
1  =>  ANSI standard C
2  =>  K&R  C with common extensions
3  =>  ANSI C with common extensions
4  =>  AT&T C++ (cfront 3.0)
5  =>  Zortech C++
6  =>  Borland C++
7  =>  Microsoft C++
8  =>  IBM Visual Age C++
9  =>  MetroWerks Code Warrior C++
10  =>  VAX and HP/Apollo C
11  =>  MetaWare High C
```

*THE DEFAULT IS K3 (ANSI C / common extensions) *

If this option is not specified, then CodeCheck will assume that the source code is ANSI C with extensions (-K3). If option -K is specified with no digit following, then CodeCheck will assume that the user meant -K0, i.e. strict K&R (1978) C with no extensions.

-L Make a listing file for the source file or project, with CodeCheck messages interspersed at appropriate points in the listing. The name of the listing file should be given immediately after the -L:

```
        check  -Lmodule.lst  module.c
```

If no name is specified, CodeCheck will use the name “check.lst”. The listing file will be created in the current directory, unless a target directory is specified with the -Q option.

-M List all macro expansions in the listing file. Each line containing a macro is first listed as it is found in the source file, and then listed a second time with all macros expanded. The -L option is redundant if -M is specified. If -L is found without -M, then the listing file created by Code Check will not exhibit macro expansions.

-N Allow nested /* comments */.

-NEST Allow C++ nested classes. When this option is in effect every union,struct, or class definition constitutes a true scope that can contain nested tag definitions. Options -K5, -K6 and -K7 imply -NEST, but -K4 does not. Use -K4 and -NEST if your C++ compiler is based on AT&T C++ version 3.0. DO *NOT* use -NEST if your C++ compiler is based on any version of C++ earlier than AT&T 3.0.

-O Append all CodeCheck stderr output to the file stderr.out. This is useful for those operating systems (e.g. MS-DOS) that do not permit any redirection or piping of stderr output.

-P Show progress of code checking. When this option is given, CodeCheck will identify each file in the project as it is opened.

-Q Specify an output directory. The pathname for the directory must follow immediately, e.g.

```
      check -Qusr/myoutput
```

When this option is specified, CodeCheck will create all of its output files in the given directory. These output files include the prototype, listing, and rule object files.

-R

-R Specify a rule file. The name of the rule file must follow immediately, e.g. check -Rmyrules mysource.c. The extension ".cc" on the rule file should be omitted. CodeCheck will look for an up-to-date object fileof the given name and extension ".cco". If this is not found, then CodeCheck will recompile and use the rule file of the given name.

-S0 Read but do not apply rules to any header files. <=== DEFAULT
-S1 Apply rules to header files given in double quotes.
-S2 Apply rules to header files given in angle brackets.
-S3 Apply rules to ALL header files.

-SQL Enable embedded SQL statements.

-T Create a file of prototypes for all functions defined in a project. The name of the prototype file should be given immediately after the -T:

```
      check -Tprotos.h source.c
```

If no name is specified, CodeCheck will use the name "myprotos.h". The prototype file will be created in the current directory, unless a target directory is specified with the -Q option.

-U Undefine a macro constant. The name of the macro must follow immediately. Thus check -UMSDOS src.c has the effect of treating src.c as though it contained the preprocessor directive #undef MSDOS.

-V Available for users. May be followed by an integer or a name.

-W Available for users. May be followed by an integer or a name.

-X Available for users. May be followed by an integer or a name.

-Y Available for users. May be followed by an integer or a name.

-Z Suppress cross-module checking. Macro definitions and variable and function declarations will not be checked for consistency across the modules of a project.

File Name Conventions

The conventions used by CodeCheck for filename extensions are:

.cc	A CodeCheck rule file, containing a set of rules for compilation by CodeCheck. These rules are written in a subset of the C language. CodeCheck requires that this extension be used for rule filenames, though it may be omitted in the **-R** command-line option.
.cch	A CodeCheck header file, for inclusion in a CodeCheck rule file.
.cco	A CodeCheck object file, produced by the CodeCheck compiler. This file contains a compilation of the rules found in the rule file with the same name but extension .cc.
.ccp	A project file for CodeCheck. This file contains a simple list of the filenames of all of the source modules that comprise a project, one filename per line. Header files and libraries should not be listed in this file.

Depending on command line options, the following optional files may be created by CodeCheck:

check.lst	The default filename for the listing file (**-L** option).
myprotos.h	The default filename for the prototype file (**-T** option).
stderr.out	The filename for stderr output (**-O** option).
temp.cco	The object file created by CodeCheck when more than one rule file is specified (**-R** option).
Default.cco	If found this compiled rule file will be used, by default.

Variables and Functions

An alphabetized master list of all CodeCheck variables, triggers, and functions() follows. See the glossary at the end of this quick reference card for definitions of terms used, or see the CodeCheck Reference manual for detailed descriptions. Variables contain information. Triggers activate conditional 'if' statements. All functions that return a pointer (*) are marked.

all_digit() 1 if a string consists of only digits.
all_lower() 1 if a string consists of only lowercase letters.
all_upper() 1 if a string consists of only uppercase letters.
atof() The standard ANSI atof function.
atoi() The standard ANSI atoi function.
class_name() Name of current C++ class or struct.
***cnv_any_to_bitfield** 1 if anything is implicitly converted to a bitfield.
***cnv_any_to_ptr** 1 if a non-pointer is implicitly converted to a pointer.
***cnv_bitfield_to_any** 1 if a bitfield is implicitly converted to anything.
***cnv_const_to_ptr** 1 if a const type is implicitly converted to a non-const.
***cnv_float_to_int** 1 if a float is implicitly converted to an integer.
***cnv_int_tofloat** 1 if an integer is implicitly converted to a float.
***cnv_ptr_to_ptr** 1 if a pointer is implicitly converted to a pointer.
***cnv_signed_to_any** 1 if a signed integer is implicitly converted to unsigned.
***cnv_truncate** 1 if an integer or float is implicitly truncated.
conflict_file() File in which conflicting

	definition occured. Valid ONLY when dcl_conflict or pp_macro_conflict is triggered.
conflict_line	Line on which conflicting definition occured. Valid ONLY when dcl_conflict or pp_macro_conflict is triggered.
corr(x,y)	Float correlation between statistics x and y.
***dcl_3dots**	1 when an ellipsis (...) is found in a declaration.
***dcl_abstract**	1 when an abstract declarator is encountered.
dcl_access	0 when a C++ member has public access, 1 when a C++ member has protected access, 2 when a C++ member has private access.
***dcl_aggr**	1 when an aggregate type is declared.
***dcl_all_upper**	1 when a declarator name is all uppercase.
***dcl_ambig**	See CodeCheck Reference Manual.
***dcl_any_upper**	1 when a declarator name has any uppercase letters.
dcl_array_dim()	If the specified level of this declarator is an array,then this function returns the array dimension (-1 if no size is given).
dcl_array_size	Total size of a declared array, -1 if no size is given, product of dimensions if the array is multidimensional.
***dcl_auto_init**	1 when an auto variable is initialized.
dcl_base	Base type of the declaration. For values see manifest constant section.
dcl_base_root	Type from which the type of dcl_base is derived from. If the type of dcl_base is not a user-defined type, dcl_base_root has same value as dcl_base.
dcl_base_name()	The base type of the current declarator, as a string.
dcl_base_name_root()	The name of type from which type of

dcl_base_name is derived. If the type of dcl_base_name is not a user-defined type, dcl_base_name_root() returns the same value as dcl_base_name().

***dcl_bitfield** 1 when a bitfield is declared.
***dcl_bitfield_anon** 1 when a bitfield has no name.
***dcl_bitfield_arith** 1 when a bitfield width requires arithmetic calculation.
dcl_bitfield_size Size in bits of the specified bitfield.
***dcl_conflict** 1 when an identifier was declared differently elsewhere. Use conflict_file() and conflict_line for location.
dcl_count Index of declarator within the current declaration list.
***dcl_cv_modifier** 1 when const or volatile is used as a non_ANSI modifier.
***dcl_definition** 1 when a declaration is a definition, not a reference.
dcl_empty 1 when an empty declaration is found (no declarator).
***dcl_enum** 1 when an enumerated constant is found.
***dcl_enum_hidden** 1 when a declarator name hides an enumerated constant.
dcl_explicit 1 when a declarator has specifier "explicit".
***dcl_extern** 1 when "extern" is explicitly specified.
***dcl_extern_ambig** See documentation.
dcl_first_upper Number of initial uppercase letters in declarator name.
***dcl_friend** 1 when a C++ friend is declared.
***dcl_from_macro** 1 when declarator name is derived from a macro expansion.
***dcl_function** 1 when a function or function typedef name is declared.
dcl_function_flags Inclusive OR of the following conditions:
1 when this function is inline, (C++)
2 when this function is virtual,

```
(C++)
                      4 when this function is pure,
(C++)
                      8 when this function is pascal,
(DOS, OS/2, Mac)
                     16 when this function is cdecl,
(DOS & OS/2)
                     32 when this function is
interrupt,(DOS & OS/2)
                     64 when this function is loadds,
(DOS & OS/2)
                    128 when this function is saveregs,
(DOS & OS/2)
                    256 when this function is fastcall.
(DOS & OS/2)
*dcl_function_ptr    1 when a pointer to a function is
      declared.
*dcl_global          1 when a variable or function has
      file scope.
*dcl_hidden          1 when a local identifier hides
another identifier.
*dcl_Hungarian       1 when a declarator name uses the
      Hungarian convention.
dcl_ident_length    Number of characters in declared
identifier name.
*dcl_init_arith      1 when an initializer uses
arithmetic.
*dcl_initializer     1 when an initializer is found.
*dcl_inline          1 when a C++ function is inline.
*dcl_label_overload  1 when a declarator name matches a
label name.
dcl_level()         See documentation.
dcl_level_flags()   See documentation.
dcl_levels          See documentation.
*dcl_local           1 when a local identifier is
declared.
*dcl_long_float      1 when a variable is declared
“long float”.
dcl_member          1 a union member identifier
                    2 a struct member identifier
                    3 a class member identifier
```

dcl_mutable	1 when an indentifier is declared 'mutable'.
dcl_name()	Current declarator name.
***dcl_need_3dots**	1 when a parameter list concludes with a comma.
***dcl_no_prototype**	1 when a function definition has no prototype in scope.
***dcl_no_specifier**	1 when a declaration has no type specifiers at all.
***dcl_not_declared**	1 when an old-style function parameter is not declared.
***dcl_oldstyle**	1 when an old-style (unprototyped) function is declared.
dcl_parameter	Index of function parameter (1 for first, etc.).
dcl_parm_count	Number of formal parameters in a function definition.
***dcl_parm_hidden**	1 if a function parameter is hidden by a local variable.
***dcl_pure**	1 when a C++ pure member function is declared.
dcl_scope_name()	scope name of current declarator.
***dcl_simple**	1 when simple variable (not pointer or array) is declared.
***dcl_signed**	1 when the “signed” type specifier is explicitly used.
***dcl_static**	1 when a declarator is static.
***dcl_storage_first**	1 when a storage class specifier is preceded by a type specifier in a declaration.
dcl_storage_flags	Set to an integer which identifies the storage class. See manifest constant section.
***dcl_tag_def**	1 when a tag is defined as part of a type specifier.
dcl_template	Number of C++ function template parameters.
***dcl_type_before**	1 when the return type of a function definition is on the line BEFORE the line with the function name.
***dcl_typedef**	1 when a typedef name is declared.
***dcl_typedef_dup**	1 when a duplicate typedef name is declared.

dcl_underscore	Number of leading underscores in declarator name.
***dcl_union_bits**	1 when a bitfield is declared as a member of a union.
***dcl_union_init**	1 when a union has an initializer.
***dcl_unsigned**	1 when a declarator is unsigned.
***dcl_variable**	1 when a variable (not a function) is declared.
***dcl_virtual**	1 when a member function is declared virtual.
dcl_zero_array	1 when an array has zero length.
define(name,body)	Define a macro with given name and body. Both the name and body must be strings. The macro may not have arguments.
eprintf()	the same as function fprintf except output to *stderr.*
exit(n)	Quit CodeCheck with return value n.
***exp_empty_initializer**	1 when an empty initializer
***exp_not_ansi**	1 when a non-ANSI expression is found.
exp_operands	Number of operands in the current expression.
exp_operators	Number of operators in the current expression.
exp_tokens	Number of tokens in the current expression.
err_message()	Returns the message body of warning message numbered as CXXXX.
err_syntax	Set to an integer when CodeCheck encounters a syntax error which is CXXXX. The value of the integer is 1 greater than the value XXXX.
fatal(n,str)	Issue fatal error #n with message str.
fclose()	CodeCheck version of the standard C function fclose.
fcn_aggr	* Number of local aggregate variables declared in function.
fcn_array	* Total number of local array elements declared in function.
***fcn_begin**	1 when a function definition begins (open brace).

fcn_members

fcn_com_lines	* Number of pure comment lines within a function.
***fcn_end**	1 at the end of function definition (close brace).
fcn_decisions	* Number of binary decision points in a function.
fcn_exec_lines	* Number of lines in function with executable code.
fcn_H_operands	* Number of Halstead operands in a function.
fcn_H_operators	* Number of Halstead operators in function.
fcn_high	* Number of high-level statements in a function.
fcn_locals	* Number of local variables declared in a function.
fcn_low	* Number of low-level statements in a function.
fcn_members	* Number of local union, struct & class members in function.
***fcn_no_header**	1 when a function definition has no comment block.
fcn_name()	Name of current function.
fcn_nonexec	* Number of non-executable statements in a function.
fcn_operands	* Number of operands in a function.
fcn_operators	* Number of operators in a function.
fcn_register	Number of register variables declared in a function.
fcn_simple	* Number of local simple variables declared in a function.
fcn_tokens	* Number of tokens found in a function.
fcn_total_lines	* Number of lines in the function definition.
fcn_u_operands	* Number of unique operands in a function.
fcn_u_operators	* Number of unique operators in a function.
fcn_uH_operands	* Number of unique Halstead operands in a function.

fcn_uH_operators	* Number of unique Halstead operators in a function.
fcn_unused	* Number of unused variables in a function.
fcn_white_lines	* Number of lines of whitespace in a function.
file_name()	Name of the current source or header file.
fopen()	Standard C function fopen.
force_include()	Specify a file to be included as header file at the beginning of each module.
fprintf()	Standard C function fprintf.
fscanf()	Standard C function fscanf.
header_name()	Name of the header that is about to be #included.
header_path()	Path to the header that is about to be #included.
histogram()	See documentation.
idn_array_dim()	If the specified level of this identifier is an array,then this function returns the array dimension (-1 if no size is given).
idn_base	Set to the base type of the identifier. See manifest constant section.
idn_base_name()	The base type of the identifier, as a string.
***idn_bitfield**	1 if the identifier is a bitfield.
***idn_constant**	1 if this identifier is an enum constant.
idn_filename()	The file in which the identifier was declared.
***idn_function**	1 if this identifier is a function name.
***idn_global**	1 if this identifier has file scope and external linkage.
idn_level()	See TechNote #14 and manual.
idn_level_flags()	See TechNote #14.
idn_levels	See TechNote #14.
idn_line	Set to the line number within the file in which this identifier was declared.
***idn_local**	1 if this identifier has local scope.

***idn_member** 1 if this identifier has class scope.

idn_name() The name of the identifier, as a string.

***idn_no_prototype** 1 if this is a function call with no prototype.

***idn_not_declared** 1 if this is a function call with no declaration.

***idn_parameter** 1 if this identifier is a function parameter.

idn_storage_flags Set to an integer which identifies the storage class of the identifier. For values of the flags, see manifest constant section.

***idn_variable** 1 if this identifier is a variable.

identifier(name) Triggers whenever the named identifier is used.

ignore(name) Instructs CodeCheck to ignore the named token.

***included(filename)** 1 if the argument header file has been included.

***isalpha(int)** 1 if the argument is an alphabetic character (a-z or A-Z).

***isdigit(int)** 1 if the argument is a decimal digit character (0-9).

***islower(int)** 1 if the argument is a lowercase alphabetic character.

***isupper(int)** 1 if the argument is an uppercase alphabetic character.

***keyword(name)** Triggers whenever the named keyword is used.

lex_ansi_escape Set to ‘a’, ‘v’, or ‘?’, respectively, when \a, \v, or \? is found within a string or character literal.

***lex_assembler** 1 when assembler code is detected.

***lex_backslash** 1 when a line is continued with a backslash character.

lex_bad_call Difference between number of actual arguments and number of formal arguments when a macro function is expanded.

lex_big_octal 8 when the digit 8 is found in an

	octal constant,
	9 when the digit 8 is found in an octal constant.
lex_c_comment	1 when comment is C /* */
lex_char_empty	1 when the empty character constant is found ('').
lex_char_long	1 when a character constant is longer than one character.
lex_constant	1 when an enumerated constant
	2 when a character constant
	3 when an integer constant
	4 when a float constant is found,
	5 when a string constant is found.
lex_cpp_comment	1 when comment is C++ //
lex_enum_comma	1 when a list of enumerated constants ends with a comma.
lex_float	1 when a numeric constant has the suffix f or F.
lex_hex_escape	Set to the number of hex digits read when a hexadecimal escape sequence (e.g. '\x1A') is found.
lex_initializer	1 when an initializer is the integer zero,
	2 when an initializer is a nonzero integer,
	3 when an initializer is a character literal,
	4 when an initializer is a float or double constant,
	5 when an initializer is a string, and
	6 when an initializer is anything else.
lex_intrinsic	1 when an intrinsic (built-in) function is called.
lex_invisible	1 when a C++ nested tag name is used without a scope.
lex_key_no_space	1 when certain keywords are not followed by whitespace.
lex_keyword	1 when the current token is a reserved keyword.

lex_lc_long	1 when a numeric constant has suffix lowercase el
lex_long_float	1 when a float constant has suffix L or l.
lex_macro	1 when a macro is about to expand.
lex_macro_token	1 when a token originates from a macro expansion.
lex_metaware	1 when any Metaware lexical extension is found.
lex_nested_comment	1 when a /*..*/ comment is found nested within another.
lex_nl_eof	1 when a nonempty source file does not end with a newline.
lex_nonstandard	1 when a character not in the standard C set is found.
lex_not_KR_escape	1 when an escape character is not in the K&R (1978) set.
lex_not_manifest	1 when a number other than 0 or 1 is not a macro.
lex_null_arg	1 when an argument is omitted from a macro function call.
lex_num_escape	Set to the numeric value when an escape sequence is found.
lex_punct_after	1 when a comma or semicolon is not followed by whitespace.
lex_punct_before	1 when a comma or semicolon is preceded by whitespace.
lex_radix	Radix of an integer constant (2, 8, 10, or 16).
lex_str_concat	1 when two strings are separated only by whitespace.
lex_str_length	Length of a string literal (not counting terminal zero).
lex_str_macro	1 when a macro name is found within a string literal.
lex_str_trigraph	1 when a trigraph is found within a string literal.
lex_suffix	1 when a numeric constant has a letter suffix.
lex_token	Index of the token in the current line (1 = first token).

***lex_trigraph**	1 when an ANSI trigraph is found.
***lex_unsigned**	1 when a numeric constant has the U or u suffix.
***lex_wide**	1 when a string or character constant has the L prefix.
lex_zero_escape	1 when an escape sequence in a character literal is zero, 2 when the escape sequence is in a string literal.
***lin_continuation**	1 when an expression is continued from the previous line.
***lin_continues**	1 when an expression is continued on the next line.
lin_dcl_count	Number of declarator names on the current line.
lin_depth	Depth of #include file nesting for the current line.
***lin_end**	1 when the end of a line is found.
***lin_has_code**	1 when a line contains code of any sort.
***lin_has_comment**	1 when a line contains a nonempty comment material.
lin_has_label	1 when a line contains a label.
lin_include_kind	1 if the line includes a project header by #include. 2 if the line includes a system header by #include.
lin_include_name()	Name of the header file included in this line.
lin_header	1 if the line comes from a project header, 2 if it comes from a system header.
lin_indent_space	Number of spaces before the first nonwhite character.
lin_indent_tab	Number of tabs before the first nonwhite character.
lin_is_comment	1 when a line contains only comment material.
lin_is_exec	1 when a line contains executable code.
lin_is_white	1 when a line is only whitespace or

empty comment.

lin_length	Length of the line in characters, not counting newline.
lin_nest_level	The statement nesting (indentation) level. See option -B.
lin_nested_comment	1 when a /*..*/ comment is found nested within another.
lin_new_comment	1 when a // comment is found.
lin_number	Index of the current line within the current file.
lin_operands	Number of operands found on the current line.
lin_operators	Number of operators found on the current line.
lin_preprocessor	1 if the current line begins with #.
lin_source	1 if it is not from a header file.
lin_suppressed	1 if it is suppressed by the preprocessor.
lin_tokens	Number of tokens on the current line.
lin_within_class	1 when the current line is within a class definition, 2 when it is in a member function but outside the class.
***lin_within_function**	1 if the current line is within a function definition.
lin_within_tag	1 if the current line is within an enumeration, 2 if it is within a union 3 if it is within a struct 4 if it is within a class
line()	The current line (as far as it has been parsed).
log2()	The logarithm base 2 of the argument.
macro(name)	Triggers when the specified macro is about to be expanded.
***macro_defined()**	1 if a specified macro has been defined.
maximum(x)	The maximum value of a statistical

	variable.
mean(x)	The mean of a statistical variable.
median(x)	The median of a statistical variable.
minimum(x)	The minimum value of a statistical variable.
mod_aggr	* Number of global array, union, struct, or class variables.
mod_array	* Number of global array elements declared in a module.
***mod_begin**	Triggers at the beginning of a module.
mod_class_lines()	Total number of lines in a classes, structs, and unions defined in a module, including member function lines.
mod_class_name()	Name of each class, struct, or union defined in a module.
mod_class_tokens()	Total number of tokens used in class, struct, and union definitions in a module, including member function tokens.
mod_classes	Number of named classes, structs, & unions defined in a module (includes template classes).
mod_com_lines	* Number of nonempty comment lines in a module.
mod_decisions	* Number of binary decision points in a module.
***mod_end**	Triggers at the end of a module.
mod_exec_lines	* Number of lines in module with executable code.
mod_extern	* Number of global variables declared with extern keyword.
mod_functions	* Number of functions defined in a module.
mod_globals	* Number of global variables declared in a module.
mod_H_operands	* Number of Halstead operands in a module.
mod_H_operators	* Number of Halstead operators in a module.
mod_high	* Number of high-level statements found in a module.
mod_low	* Number of low-level statements

found in a module.

mod_macros		Number of macros defined in a module.
mod_members	*	Number of union, struct, or class members declared.
mod_name()		Name of the current module.
mod_nonexec	*	Number of non-executable statements in a module.
mod_operands	*	Total number of operands used in a module.
mod_operators	*	Total number of operators used in a module.
mod_simple	*	Number of local simple variables defined in a module.
mod_static	*	Number of static global variables defined in a module.
mod_tokens	*	Number of tokens found in a module.
mod_total_lines	*	Total number of lines in a module.
mod_u_operands	*	Number of unique operands used in a module.
mod_u_operators	*	Number of unique operators used in a module.
mod_uH_operands	*	Number of unique Halstead operands in a module.
mod_uH_operators	*	Number of unique Halstead operators in a module.
mod_unused	*	Number of static global variables declared but not used.
mod_warnings		Number of warnings issued by CodeCheck for a module.
mod_white_lines	*	Number of white and empty comment lines in a module.
mode(x)		The mode (most common value) of a statistical variable.
ncases(x)		The number of cases recorded in a statistical variable.
next_char()		The lookahead character at the currently parsed position.
new_type()		Create new intrinsic type specifiers. See reference manual.
no_undef(name)		1 if the argument has not been

previously #undefined.

All following op_ variables are triggers.

op_add + the binary addition operator (NOT the unary plus).
op_add_assign += the add-assign operator.
op_address & the address-of operator.
op_and_assign &= the bitwise-and-assign operator.
op_array_dim() If the specified level of the specified operand is an array, then this function returns the array dimension (-1 if no size is given).
op_arrow -> the indirect member selector operator.
op_assign = the assignment operator.
op_assoc => the Metaware association-operator.
op_base() See TechNote #14.
op_base_name() See TechNote #14 and manual
op_based :> the Microsoft based operator.
op_bit_and & the bitwise-and operator.
op_bit_not ~ the bitwise-complement operator.

op_bit_or | the bitwise-inclusive-or operator.
op_bit_xor ^ the bitwise-exclusive-or operator.
op_bitfield(j) 1 if operand j is a bitfield.
op_bitwise Any bitwise operator is used.
op_break The “break” keyword.
op_call The function-call operator.
op_cast Any cast operator (including C++ function-like casts).
op_cast_to_ptr A cast-to-pointer in the form (Type *).
op_catch Trigger on the "catch" keyword.
op_close_angle > the right angle bracket, used as a C++ template delimiter.
op_close_brace } the right curly brace.

op_close_bracket	]	the right square bracket.
op_close_funargs	)	the end-argument-list parenthesis.
op_close_paren	)	the right parenthesis.
op_close_subscript	]	the end-of-subscript operator.
op_colon_1	:	the unary colon (e.g. after a label).
op_colon_2	:	the binary colon (e.g. in a conditional expression).
op_comma	,	the comma operator (NOT the comma separator).
op_cond	?:	the conditional operator.
op_continue		The "continue" keyword.
op_declarator		Any operator found within a declaration.
op_delete		The C++ delete operator.
op_destroy	~	the C++ destructor symbol.
op_div	/	the division operator.
op_div_assign	/=	the divide-assign operator.
op_do		The "do" keyword.
op_else		The "else" keyword
op_equal	==	the equality-test operator.
op_executable		Any operator found within executable code.
op_for		The "for" keyword.
op_function()		The name of a function called or declared.
op_goto		The "goto" keyword.
op_high		Any high-precedence operator.
op_if		The "if" keyword.
op_indirect	*	the indirection operator (NOT the declarator symbol).
op_infix		Any infix operator.
op_init	=	the initialization operator.
op_iterator	->	the Metaware iterator-definition operator.
op_iterator_call	<-	the Metaware iterator-call operator.
op_keyword		Any executable keyword.
op_left_assign	<<=	the shift-left-assign operator.
op_left_shift	<<	the shift-left operator.

op_less	<	the less-than operator.
op_less_eq	<=	the less-than-or-equal-to operator.
op_level()		See TechNote #14 and manual.
op_level_flags()		See TechNote #14.
op_levels()		See TechNote #14.
op_log_and	&&	the logical-and operator.
op_log_not	!	the logical-negation operator.
op_log_or	||	the logical-or operator.
op_low		Any low-precedence operator.
op_macro()		The name of the macro function about to be expanded.
op_macro_call	(	the macro-function-expand operator.
op_medium		Any operator that is neither low- nor high-precedence.
op_member	.	the member-of operator.
op_memptr	->*	the C++ member-pointer operator.
op_memsel	.*	the C++ member-selector operator.
op_more	>	the greater-than operator.
op_more_eq	>=	the greater-than-or-equal-to operator.
op_mul	*	the multiplication operator.
op_mul_assign	*=	the multiply-assign operator.
op_negate	-	the unary negation operator (NOT subtraction).
op_new		The C++ new operator.
op_not_eq	!=	the not-equal-to operator.
op_open_angle	<	the left angle bracket, used as a C++ template delimiter.
op_open_brace	{	the left curly brace.
op_open_bracket	[	the left square bracket.
op_open_funargs	(	the function-argument-list parenthesis. Use op_declarator to determine whether the context is a function declaration or a function call.
op_open_paren	(	the left parenthesis.
op_operands		The number of operands used by an executable operator.
op_or_assign	|=	the bitwise-or-assign operator.
op_parened_operand()		1 if the specified operand is in

parentheses.

op_plus	+	the unary plus operator (NOT addition).
op_pointer	*	the pointer-to declaration operator (NOT indirection).
op_post_decr	—	the post-decrement operator.
op_post_incr	++	the post-increment operator.
op_postfix		Any postfix operaotr.
op_pre_decr	—	the pre-decrement operator.
op_pre_incr	++	the pre-increment operator.
op_prefix		Any prefix operator.
op_punct		Any punctuation operator.
op_reference	&	the C++ reference-to declaration operator.
op_rem	%	the remainder operator.
op_rem_assign	%=	the remainder-assign operator.
op_return		The “return” keyword.
op_right_assign	>>=	the right-shift-assign operator.
op_right_shift	>>	the right-shift operator.
op_scope	::	the C++ scope operator.
op_semicolon	;	the semicolon.
op_separator	,	the comma separator (NOT the comma operator).
op_sizeof		The sizeof operator.
op_space_after		An operator is followed by a space character.
op_space_before		An operator is preceded by a space character.
op_sub_assign	-=	the subtract-assign operator.
op_subscript		the subscript operator.
op_subt	-	the binary subtraction operator (NOT unary negation).
op_switch		The “switch” keyword.
op_throw		Trigger on the "throw" keyword.
op_try		The "try" keyword.
op_while_1		The “while” keyword (unless part of do-while).
op_while_2		The “while” keyword when used with “do”.
op_white_after		An operator is followed by

whitespace.

op_white_before	An operator is preceded by whitespace.
op_xor_assign	^= the exclusive-or-assign operator.
option(char c)	1 if the command-line option -c is in effect

The previous op_ variables were triggers.

pow(x,y)	Standard ANSI C pow function.
pp_ansi	1 whenever a new ANSI preprocessor feature is encountered.
pp_arg_count	Number of formal parameters in a macro definition.
pp_arg_multiple	1 if a formal parameter is used more than once.
pp_arg_paren	1 if a formal parameter is not enclosed in parentheses.
pp_arg_string	1 if a formal parameter is found within a string.
pp_arith	1 if a conditional requires an arithmetic calculation.
pp_assign	1 if a macro definition is a simple assignment.
pp_bad_white	1 if a whitespace character is neither a space nor a tab.
pp_benign	1 if a macro is redefined equivalently.
pp_comment	1 if two tokens in a macro are separated by a comment.
pp_const	1 if a macro is a manifest constant.
pp_defined	1 if the “defined” preprocessor function is found.
pp_depend	1 if #undef is used on a macro required by another macro.
pp_elif	1 if the #elif directive is found.
pp_empty_arglist	1 if a macro function definition has no parameters.
pp_empty_body	1 if the definition of a macro has

no body.

pp_endif 1 if the #endif directive is found.

pp_error 1 if the #error directive is found.

pp_error_severity() Control the leniency of #error directives - Fatal or Informational.

pp_if_depth Depth whenever a conditional (e.g. #if) is activated.

pp_include 1 if #include pathname is in “”, from a macro expansion,
2 if #include pathname is in “”, not from a macro,
3 if #include pathname is in <>, from a macro expansion,
4 if #include pathname is in <>, not from a macro,
5 if #include pathname is not enclosed (Metaware only).
6 if #include filename is not enclosed (Vax VMS only).

pp_include_depth Depth of inclusion when an #include is performed.

***pp_include_white** 1 if pathname in an #include has leading whitespace.

***pp_keyword** 1 if a macro name is a reserved ANSI or C++ keyword.

***pp_length** Length in characters of macro body (excluding whitespace).

***pp_lowercase** 1 if a macro name has any lowercase letters.

***pp_macro** Length in characters of a macro name.

***pp_macro_conflict** 1 when a macro was defined differently elsewhere. Use conflict_file() and conflict_line for location.

***pp_macro_dup** 1 if a macro is defined in more than one file.

pp_name() Name of the macro currently being defined.

***pp_not_ansi** 1 if any non-ANSI preprocessor usage is found.

***pp_not_defined** 1 if a conditional uses an undefined identifier.

***pp_not_found**	1 if an #include file could not be found.
pp_overload	1 if a declared identifier matches a macro function name.
pp_paste	1 if the ANSI paste operator (##) is found.
pp_paste_failed	1 if a the operands for ## could not be pasted together.
pp_pragma	1 if a #pragma directive is found.
pp_recursive	1 if a recursive macro definition is found.
pp_relative	1 if an #include in a header file uses a relative pathname.
pp_semicolon	1 if a macro definition ends with a semicolon.
pp_sizeof	1 if a directive requires evaluating a “sizeof”.
pp_stack	1 if a macro is redefined within a module (except benign).
pp_stringize	1 if the ANSI stringize operator (#) is found.
pp_sub_keyword	1 if a directive name is itself a macro name.
pp_trailer	1 if a directive line ends with any nonwhite characters.
pp_undef	1 if an #undef directive is found.
pp_unknown	1 if a directive unknown to CodeCheck is found.
pp_unstack	1 if an #undef is used to unstack multiply-defined macros.
pp_white_after	Length of whitespace that precedes the # character.
pp_white_before	Length of whitespace that follows the # character.
pragma()	Triggers when the specified pragma is encountered.
prefix()	See documentation.
prev_token()	The previous lexical token (as a string).
printf()	The standard ANSI printf function.
prj_aggr	Number of external array, union, struct, class variables.

prj_array	Number of external array elements in a project.
prj_begin	Triggers at the beginning of a project.
prj_com_lines	Number of nonempty comment lines in a project.
prj_conflicts	Number of conflicting macro definitions in a project.
prj_decisions	Number of binary decision points in a project.
prj_end	Triggers at the end of a project.
prj_exec_lines	Number of line in project with executable code.
prj_functions	Number of functions defined in a project.
prj_globals	Number of external variables defined in a project.
prj_H_operands	Number of Halstead operands in a project.
prj_H_operators	Number of Halstead operators in a project.
prj_headers	Number of distinct header files read in a project.
prj_high	Number of high-level statements found in a project.
prj_low	Number of low-level statements found in a project.
prj_macros	Number of distinct macros defined in a project.
prj_members	Number of external union, struct, or class members.
prj_modules	Number of source modules in a project.
prj_name()	Name of the current project file
prj_nonexec	Number of non-executable statements in a project.
prj_operands	Number of operands found in a project.
prj_operators	Number of operators found in a project.
prj_simple	Number of external global variables

	defined in a project.
prj_tokens	Number of lexical tokens found in a project.
prj_total_lines	Number of lines in a project.
prj_u_operands	Number of unique operands in a project.
prj_u_operators	Number of unique operators in a project.
prj_uH_operands	Number of unique Halstead operands in a project.
prj_uH_operators	Number of unique Halstead operators in a project.
prj_unused	Number of unused external variables in a project.
prj_warnings	Number of CodeCheck warnings issued for a project.
prj_white_lines	Number of white and empty comment lines in a project.
quantile()	Returns the specified quantile of a statistical variable.
remove_path()	Remove the least recently set including path from searching list.
reset()	Deletes all cases recorded in a statistical variable.
root()	Current declarator name after prefixes have been removed.
scanf()	Standard ANSI C scanf function.
set_header_optS()	Set option -S for specified file overriding the option -S set globally.
set_option()	Sets the specified command-line integer option.
set_str_option()	Sets the specified command-line string option.
sprintf()	The standard ANSI sprintf function.
skip_macro_ops()	Control if op_variables applicable on operators derived from macro expansion.
skip_nonansi_indent()	Control if ignore identifier starting with characters '@', '$' or '`'.
sqrt()	Standard ANSI C square-root function.
sscanf()	The ANSI stdlib sscanf() function.
stdev()	Standard deviation of a statistical

variable.
stm_aggr Number of array, union, struct, class variables declared.
stm_array Number of local array elements declared.
***stm_bad_label** 1 if a label is not attached to any statement.
stm_cases Number of case or default labels on this statement.
stm_catchs Number of handlers (catches) in a try-block.
stm_container Set to a value which indicates the kind of high-level statement that contains the current statement. See stm_kind (below) for the possible values.
stm_cp_assign Number of compound assignment operators.
stm_cp_begin At the open curly brace of a compound statement, this variable is set to a value that indicates the kind of statement that contains the compound statement. See stm_kind (below) for the possible values.
stm_depth Nesting depth of a statement within other statements.
***stm_end** Triggers at the end of any statement.
***stm_end_tryblock** 1 if the closing brace is found of the last catch of a try-block.
***stm_goto** 1 if a goto enters a block with auto initializers.
***stm_if_else** 1 if an if statement has a matching else statement.
***stm_is_comp** Set to the same value as stm_cp_begin, at the END of a compound statement (the close curly brace).
***stm_is_expr** 1 if a statement is an expression.
***stm_is_high** 1 if a statement is compound, selection, or iteration.
***stm_is_iter** 1 if a statement is a for, while, or do-while.
***stm_is_jump** 1 if a statement is a goto,

```
continue, break, or return.
*stm_is_low           1 if a statement is an expression
or jump statement.
*stm_is_nonexec       1 if a statement is not executable
(i.e. a declaration).
*stm_is_select        1 if a statement is an if, if-
else, or switch.
stm_kind             1 for an "if" statement,
                     2 for an "else" statement,
                     3 for a "while" statement,
                     4 for a "do" statement,
                     5 for a "for" statement,
                     6 for a "switch" statement,
                     7 for a "function" compound
statement,
                     8 for a compound statement,
                     9 for an expression statement,
                    10 for a break statement,
                    11 for a continue statement,
                    12 for a return statement,
                    13 for a goto statement,
                    14 for a declaration statement,
                    15 for an empty statement.
stm_labels           Number of ordinary labels (not case
or default labels)
                       attached to this statement.
stm_lines            Number of lines in the current
statement,including blank lines that precede the first
token of the statement.
stm_locals           Number of local variables declared
in a block.
*stm_loop_back        1 if a goto statement jumps
backward.
stm_members          Number of local union, struct, or
class members declared.
*stm_need_comp       1 if the statement contained by if,
else, for , while and do is not a compound statement.
*stm_never_caught    1 if a handler( catch ) will never
be reached.
*stm_no_break         1 if the previous statement is a
case with no jump.
*stm_no_default       1 if a switch statement has no
```

default case.
***stm_no_init** 1 if a variable is used before it has been initialized. Note: this variable does not yet work on C++ code.
stm_operands Total number of operands found in a statement.
stm_operators Total number of C operators found in a statement.
stm_relation Number of Boolean relational operators in a statement.
stm_return_paren 1 if return has a value NOT enclosed in parentheses.
stm_return_void 1 if return value conflicts with the function declaration.
stm_semicolon 1 if a suspicious semicolon is found (e.g. while(x);).
stm_simple Number of local simple variables declared in a block.
stm_switch_cases Number of cases found in the current switch statement.
stm_tokens Number of lexical tokens found in a statement.
stm_unused Number of unused local variables in a block. Use function stm_unused_name(k) for their names (0<=k<stm_unused).
stm_unused_name() Returns name of the given unused variable in the block.
strcat() Standard ANSI C strcat() function.
strchr() Standard ANSI C strchr() function.
strcmp() Standard ANSI C strcmp() function.
strcpy() Standard ANSI C strcpy() function.
strcspn() Standard ANSI C strcspn() function.
strequiv() 1 if one string is the same (except for case) as another.
strlen() Standard ANSI C strlen() function.
strncat() Standard ANSI strncat function.
strncmp() Standard ANSI strncmp function.
strncpy() Standard ANSI strncpy function.
str_option() Returns string value of the specified command-line option.
strpbrk() Standard ANSI strpbrk function.
strrchr() Standard ANSI strrchr function.

strspn() Standard ANSI strspn function.
strstr() Standard ANSI C strstr function.
suffix() Similar to the prefix function. See documentation.
***tag_abstract** 1 when this is a C++ anonymous class.
***tag_anonymous** 1 when an anonymous (unnamed) tag is defined.
***tag_base_access** 1 when a base class does not have an explicit access specifier (public, protected, or private).
tag_bases Number of C++ base classes for this tag.
tag_baseclass_access() The access specifier of a specified base class.
tag_baseclass_kind() The tag kind of a specpfied base class.

2	for a union
3	for a struct
4	for a class

tag_baseclass_name() The name of a specified base class.
***tag_begin** 1 when a tag definition begins.
tag_classes Number of named classes nested within this class.
tag_components() See documentation.
tag_constants Number of enumerated constants defined in this class.
tag_constructors Number of constructors declared in this class.
tag_distance

1	for a _near tag,(Borland C++)
2	for a _far tag, (Borland C++)
3	for a _huge tag, Borland C++)
4	for an _export tag. (Borland C++)

***tag_end** 1 when a tag definition ends.
tag_fcn_friends Number of friend functions declared in this class.
tag_friends Number of friend classes declared in this class.
tag_functions Number of member functions declared in this class.

*tag_global	1 if this tag has file scope.
*tag_has_assign	1 if this C++ class has an operator=().
tag_has_copy	1 if this C++ class has a copy constructor.
tag_has_default	1 if this C++ class has a default constructor.
tag_has_destr	1 if this C++ class has a destructor.
tag_hidden	1 when a local tag hides another tag.
tag_kind	1 for an enum, 2 for a union, 3 for a struct, 4 for a class.
tag_lines	Number of lines in the tag definition.
tag_local	1 if this tag has local scope (within a function).
tag_mem_access	1 if the first member of this class does not have an access label (public, protected, or private).
tag_members	Number of data members defined in this class.
*tag_name()	Returns the tag name for the current tag.
*tag_nested	1 if this tag definition is nested within another tag.
tag_operators	Number of operator functions declared in this class.
tag_private	Number of identifiers declared with private access.
tag_protected	Number of identifiers declared with protected access.
tag_public	Number of identifiers declared with public access.
tag_static_fcn	Number of static member functions declared in this class.
tag_static_mem	Number of static data member declared in this class.
tag_template	Number of template parameters.

tag_types

tag_tokens	Number of tokens in this tag definition.
tag_types	Number of typedef names defined in this class.
test_needed()	Triggers if any of the specified functions is called without a validity test immediately following. Normally used to verify that return value from malloc() was tested.
token()	Returns current lexical token as a string.
undefine()	Undefines the specified macro.
variance()	Variance of a statistical variable.
warn()	Generates a warning message.

CodeCheck Manifest Constants

This section defines manifest constants for the following CodeCheck variables and functions:

dcl_base
dcl_base_root
dcl_function_flags
dcl_level()
dcl_level_flags()
dcl_storage_flags
lin_header
lin_include_kind
lin_preprocessor
lin_within_tag
op_base()
op_level()
op_level_flags()
pp_error_severity()
stm_container
stm_cp_begin
stm_is_comp
stm_kind
tag_kind

The values of **lex_constant:**

```
#define     CONST_BOOL        1
#define     CONST_ENUM        2
#define     CONST_CHAR        3
#define     CONST_INTEGER     4
#define     CONST_FLOAT       5
#define     CONST_STRING      6
```

This values of **lex_initializer:**

```
#define     INIT_ZERO         1
#define     INIT_INTEGER      2
#define     INIT_BOOL         3
#define     INIT_CHAR         4
```

CodeCheck Manifest Constants

```
#define     INIT_FLOAT       5
#define     INIT_STRING      6
#define     INIT_OTHER       7
```

The declarator base types for **dcl_base, dcl_base_root,**and **op_base():**

```
#define VOID_TYPE               1
#define BOOL_TYPE               2
#define CHAR_TYPE               3
#define SHORT_TYPE              4
#define WCHAR_TYPE              5
#define INT_TYPE                6
#define LONG_TYPE               7
#define LONG_LONG_TYPE          8
#define EXTRA_INT_TYPE          9
#define UCHAR_TYPE              10    // unsigned char
#define USHORT_TYPE             11    // unsigned short
#define UINT_TYPE               12    // unsigned int
#define ULONG_TYPE              13    // unsigned long
#define EXTRA_UINT_TYPE 14    // non-standard
#define FLOAT_TYPE              15
#define SHORT_DOUBLE_TYPE       16
#define DOUBLE_TYPE             17
#define LONG_DOUBLE_TYPE        18
#define INT8_TYPE               19

// __int8, __int16, __int32 and __int64 are types of
// IBM, Borland, & Microsoft C++.

#define INT16_TYPE              20   // non-standard
#define INT32_TYPE              21
#define INT64_TYPE              22

#define EXTRA_FLOAT_TYPE        23
#define ENUM_TYPE               24
#define UNION_TYPE              25
#define STRUCT_TYPE             26
#define CLASS_TYPE              27
#define DEFINED_TYPE            28
#define EXTRA_PTR_TYPE          29
```

CodeCheck Manifest Constants

```
#define CONSTRUCTOR_TYPE      30
#define DESTRUCTOR_TYPE       31
#define TEMPLATE_TYPE 32    //  C++ template parameter

#define COMP_TYPE       EXTRA_INT_TYPE    // Macintosh
#define EXTENDED_TYPE   LONG_DOUBLE_TYPE
#define DERIVED_TYPE    DEFINED_TYPE      // Obsolete
#define SEGMENT_TYPE    EXTRA_PTR_TYPE    // Microsoft
```

The values of **dcl_function_flags:**

```
#define     INLINE_FCN             1
#define     VIRTUAL_FCN            2
#define     PURE_FCN               4
#define     PASCAL_FCN             8
#define     CDECL_FCN             16
#define     INTERRUPT_FCN         32
#define     LOADDS_FCN            64
#define     SAVEREGS_FCN         128
#define     FASTCALL_FCN         256
#define     EXPORT_FCN           512
#define     EXPLICIT_FCN        1024
```

The values of **dcl_level()** and **op_level()**

```
#define     SIMPLE                 0
#define     FUNCTION               1
#define     REFERENCE              2
#define     POINTER                3
#define     ARRAY                  4
```

The values of **dcl_level_flags()** and **op_level_flags():**

```
#define CONST_FLAG             1 // constant pointer
#define VOLATILE_FLAG          2 // volatile pointer
#define NEAR_FLAG              4
#define FAR_FLAG               8
#define HUGE_FLAG             16
#define EXPORT_FLAG           32 //  Windows only
#define BASED_FLAG            64 //  Microsoft only
#define SEGMENT_FLAG         128 // Borland, Microsoft
```

CodeCheck Manifest Constants

The values of **dcl_storage_flags:**

```
#define EXTERN_SC              1
#define STATIC_SC              2
#define TYPEDEF_SC             4
#define AUTO_SC                8
#define REGISTER_SC           16
#define MUTABLE_SC            32
#define GLOBAL_SC             64
```

The value of lin_header **and** lin_include_kind

```
#define PRJ_HEADER           1     //    Project
header (filename in quotes)
#define SYS_HEADER           2     //    System header
(filename in angle brackets)
```

Values for any of these variables:
stm_kind
stm_container
stm_is_comp
stm_cp_begin

```
#define IF        1   // if statement
#define ELSE      2   // else statement
#define WHILE     3   // while statement
#define DO        4   // do statement
#define FOR       5   // for statement
#define SWITCH    6   // switch statement
#define TRY       7   // try statement
#define CATCH     8   // catch statement
#define FCN_BODY  9   // function definition
#define COMPOUND  10   // compound statement
#define EXPRESSION 11   // expression statement
#define BREAK     12   // break statement
#define CONTINUE  13   // continue statement
#define RETURN    14   // return statement
#define GOTO      15   // goto statement
#define DECLARE   16   // declaration statement
```

```
#define EMPTY     17   // empty statement
```

The values of **tag_kind** and **lin_within_tag:**

```
#define ENUM_TAG              1
#define UNION_TAG             2
#define STRUCT_TAG            3
#define CLASS_TAG             4
```

The value to be passed into function **pp_error_severity()** as argument:

```
#define INFO_PP   0    // #error will be treated as
informative.
#define ERROR_PP  1    // #error will fatal program
exit.
```

The value of **lin_preprocessor**

```
#define DEFINE_PP_LIN       1
#define UNDEF_PP_LIN        2
#define INCLUDE_PP_LIN      3
#define IF_PP_LIN           4
#define IFDEF_PP_LIN        5
#define IFNDEF_PP_LIN       6
#define ELSE_PP_LIN         7
#define ELIF_PP_LIN         8
#define ENDIF_PP_LIN        9
#define PRAGMA_PP_LIN      10
#define LINE_PP_LIN        11
#define ERROR_PP_LIN       12
#define ASM_PP_LIN         13
#define ENDASM_PP_LIN      14
#define IMPORT_PP_LIN      15
#define CINCLUDE_PP_LIN    16
#define RINCLUDE_PP_LIN    17
#define RCINCLUDE_PP_LIN   18
#define INC_NEXT_PP_LIN    19
#define OPTION_PP_LIN      20
```

CodeCheck Manifest Constants

```
                   Predefined Constants

#define     NULL        0
#define     TRUE        1
#define     FALSE       0
```

System Dependent Constants

These constants are defined every time CodeCheck is executed.

Constant	***Value***	***Comment***
CODECHECK	801	Major Version
BETA	2	Minor Version
lint	2	
__STDC__	1	Option -k2 **only.**
__STDC__	0	**Except** option -k2.
__cplusplus	1	C++ **only** (-k4 through -k9).
cplusplus	1	C++ **only** (-k4 through -k9).
__FILE__	<file name>	
__LINE__	<line number>	
__DATE__	<date>	
__TIME__	<time>	
__builtin_va_alist	arg0	

The following constants are defined if the CodeCheck program is compiled for the operating system specified. If you wish to use CodeCheck on source code for operating systems other than the default then appropriate constants must be set explicitly.

Unix Operating System	
unix	1
__unix	1
DOS Operating System	
MSDOS	1
M_I86	1
M_I86LM	1
__I86__	3
__MSDOS__	1

System Dependent Constants (OS)

Constant	***Value***	***Comment***
__LARGE__	1	
__BORLANDC	0x500	
__TURBOC__	0x500	
_WIN32	1	
OS/2 Operating System		
__OS2__	1	
__IBMC__	200	
__FLAT__	1	
__32BIT__	1	
__386__	1	
_M_I386	1	
_WIN32	1	
NT Operating System		
_M_IX86	300	
_MSC_VER	800	
_MSDOS	1	
_X86_	1	
i386	1	
MSDOS	1	
_WIN32	1	1
VMS Operating System		
vax	1	
vms	1	
vaxc	1	
vax11c	1	
VAX	1	
VMS	1	
VAXC	1	
CC$gfloat	1	
CC$parallel	1	

System Dependent Constants (C++)

These constants are defined when options K6 through K9 are enabled.

Constant	***Value***	***Comment***
Macintosh Operating System		
applec	1	
MC68000	1	
mc68000	1	
m68k	1	
macintosh	1	
Borland C++		
__CDECL__	1	
__BCPLUSPLUS__	0x0340	
__TCPLUSPLUS__	0x0340	
__TEMPLATES__	1	
wchar_t	short	OS/2 **only.**
Microsoft C++		
__single_inheritance		Expands to nothing.
__multiple_inheritance		Expands to nothing.
__virtual_inheritance		Expands to nothing.
_M_I86	1	**Except** Windows NT.
_M_I86LM	1	**Except** Windows NT.
_M_IX86	300	
_MSC_VER	1100	
_MSDOS	1	
_X86_	300	
i386	1	
MSDOS	1	

System Dependent Constants (C++)

These constants are defined when options K6 through K9 are enabled.

Constant	***Value***	***Comment***
Metaware High C		
__HIGHC__	1	
Symantec C++		
__SC__	700	
IBM VisualAge C++		
__IBMCPP__	350	
Metrowerks CodeWarrior C++		
__MWERKS__	1	Macintosh

Glossary

Glossary of terms used in this reference guide.

abstract declarator
 - A type without a declarator name, e.g. (char **).
aggregate type
 - Array, union, struct, or class.
anonymous tag
 - An enum, union, struct, or class defined without a name.
argument of a function
 - A value actually passed to a function during a call (see parameter).
base type
 - The simple type of an identifier before any qualification. For example, the declaration "const double *xyz[5]" has base type "double".
block
 - A compound statement or function body.
compound statement
 - A block of statements enclosed in curly braces.
declarator
 - An identifier that is being declared.
definition
 - A declaration that allocates space for a variable or function, as opposed to a declaration that merely refers to a variable or function.
directive
 - A preprocessor instruction (all directives begin with #).
global
 - A variable with file scope, whether or not it is static.
Halstead operator
 - Any token that is not an identifier.
high precedence operator
 - Any of these operators:

```
        &   (address of)
        ()    (function call)
        ->    (pointer dereference)
         ~    (bitwise logical complement)
        ++    (pre- or post-increment)
         —    (pre- or post-decrement)
         *    (indirection)
         !    (logical negation)
         .    (member selection)
        ->*   (C++ member dereference)
        .*    (C++ member selection)
         -    (unary arithmetic negative)
         +    (unary arithmetic positive)
        ::    (C++ scope)
        []    (subscript)
```

iteration-statement
- A for-, while-, or do-while-statement.

jump-statement
- A goto-, continue-, break-, or return-statement.

local
- A variable with block scope, declared within a function.

low precedence operator
- Any of these operators:

        ?: (conditional)
   = += -= *= /= &= |= %= ^=

assignments)

manifest constant
- A constant referred to with a symbol rather than a value.

medium precedence operator
- Any operator not listed above as low- or high-precedence.

newline
- Depending on the system, a newline "character" may be a carriage return, a linefeed, a return followed by a linefeed, or a linefeed followed by a return. Like most compilers, CodeCheck accepts any of these.

parameter

parameter of a function
 - The name of a value received by a function in a call (see argument).
oldstyle function
 - An unprototyped function.
rule file
 - An ascii (.cc) file that contains CodeCheck expert systems rules, which are event driven. The language is a subset of C.
selection statement
 - if-statement, if-else-statement, or switch-statement.
simple type
 - a type that is NOT an array, pointer, reference, or function.
statistic type
 - A special CodeCheck storage class. Statistical variables remember every value ever assigned to them.
tag name
 - The “tag” of an enum, union, struct, or class is the identifier that immediately follows the keyword enum, union, struct, or class.
trigger
 - A CodeCheck variable which is event driven and may conditionally activate a selection statement 'if' in a CodeCheck rule file.
whitespace
 - One or more of these characters: space, tab, newline, vertical tab, form-feed, backspace. Comments within macro definitions are whitespace.