CodeCheck Reference Manual

by
Loren Cobb, PhD.

CodeCheck™ is a product of Abraxas Software, Inc.

For more information, contact:

Abraxas Software, Inc.
Post Office Box 19586
Portland, OR 97280, USA

Phone: 503-232-0540
Fax: 503-232-0543

Email: support@abxsoft.com
www.abraxas-software.com

Table of Contents

Introduction

Computer professionals have a large and growing collection of tools that aid in the program development process. Oddly, very few of these tools for programmers are themselves fully programmable. CodeCheck is a significant addition to the genre of programmable tools. It performs its primary task—analyzing and critiquing C and C++ source code—entirely under the direction of a user-written control program.

CodeCheck is not a new version of that old C programmer's standby, *lint*, although it can perform some *lint*-like error detection. For example, CodeCheck compares all declarations and macro definitions across all modules of a project, to detect inconsistencies.

The main thrust of CodeCheck is to detect noncompliance with codified style standards, to detect maintenance or portability problems within code which already compiles perfectly on today's compilers, and to compute customized quantitative indicators of code size, complexity, and density.

CodeCheck is a powerful tool for analyzing C and C++ source code. Standards and measures can be specified by the user for a tremendous number of features of C code that have an impact on *portability*, *maintainability*, and *style*. CodeCheck is designed to enhance dramatically the effectiveness and efficiency of project management in commercial and industrial programming efforts.

A custom CodeCheck program specifying code standards and measures can be written by a project leader using the CodeCheck language (actually a restricted subset of C itself). CodeCheck can be programmed to:

a. Monitor compliance with standards for programming style, rules for type-encoded prefixes for identifiers, proper use of macros and typedefs, prototypes, *etc.*

b. Identify code that is not portable to or from any particular environment (machine, compiler, operating system, or interface standard).

c. Quantify code maintainability with user-defined measures at all levels: line, statement, function, file, and project. Compute McCabe and Halstead complexity measures.

Sample CodeCheck programs are provided for a variety of problems, ranging from portability to complexity and compliance analysis for corporate and industry style standards.

Chapter 1: Command-line Options

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, one per line (*do not* list header files and libraries in the project file). Give this project file the extension ".ccp". The contents after character '#' are interpreted as comments and ignored till the end of the line. Then invoke CodeCheck, specifying the project file instead of a source file:

check -options myproject.ccp # comments

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. Command-line options may also be specified in the project file, one per line. Every option placed in a project file applies to every source file in the project.

The CodeCheck functions option(char) and set_option(char,int) can be used to obtain and set simple and integer-valued command-line options, *e.g.* **–B** and **–N**. It cannot be used to change **–K**, however. For those command-line options that take string operands, *e.g.* **–I**usr/foo/bar/headers, the equivalent functions are str_option(char) and set_str_option(char, char *). The CodeCheck function option() returns an integer whose value depends on the command-line options specified by the user when CodeCheck was invoked. For example, option('X')

returns the value 1 if the user specifies the option **–X** in the command line, otherwise it returns 0.

A user can place either an integer or a string after the option. In both cases, the value of the option can be obtained by calling function str_option('X') in rule a rule file. To use the option as an integer in a rule file, pass the value obtained via function call to str_option to another CodeCheck function atoi(). These functions are useful for three purposes, as outlined in sections 1.3 – 1.5.

1.1 Options

Command-line options are used to override default actions or conventions, or to indicate additional actions that you want CodeCheck to perform. CodeCheck command-line options are ***not*** case-sensitive. The available options are:

–A Reserved for CodeCheck expansion. Please do not use.

–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 Reserved for CodeCheck expansion. Please do not use.

–D Define a macro. The name of the macro must follow immediately. An optional macro definition can be specified after an equal sign. The macro may not have any arguments. For example,

```
check –DFOREVER=for(;;)
```

has the same effect as starting each source code file with

```
#define FOREVER for(;;)
```

If no macro definition is given, then CodeCheck assigns the value 1 to the macro by default.

-D? Internal macro dump [#define]. On command line –D? will dump initial state of #define internals after –Kn initialization Dynamic dumps may be generated with rule-file set_str_option(‘D’, “?”) function.

–E Do not ignore tokens that are derived from macro expansion when performing counts, *e.g.* of operators and operands. The default (–**E** not specified) 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 Do not read each header file more than once. *Caution:* Some header files are designed to be read multiple times, with conditional access to different sections of the header.

–H List all header files in the listing file. The **–L** option is assumed if this option is found. If **–L** is found without **–H**, then the listing file created by CodeCheck will not display the contents of header files.include files

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

```
check -I/usr/myheaderpath src.c
```

Header directory pathnames identified with the **–I** command-line option are searched *before* any directory paths listed in the the INCLUDE environmental variable. *CodeCheck Unix only:* the default header directory path is /usr/include.

–J Suppress all CodeCheck-generated error messages, *e.g.* syntax warnings. This option does ***not*** suppress warning messages generated by rules.

–K*n* Identify the dialect of C to be assumed for the source files. A digit should follow immediately, which identifies the dialect. The dialects of C and C++ currently available are:

0	Strict K&R (1978) C
1	Strict ANSI standard C
2	K&R C with common extensions
3	ANSI C with common extensions ***(default)***
4	AT&T C++ [ANSI STD C++ - Stroustrup]
5	Symantec C++
6	Borland C++ [CodeBuilder]
7	Microsoft C++ [MSDEV C++ 6.0 or later]
8	GNU-GCC C/C++ [IBM VAC++]
9	MetroWerks CodeWarrior C/C++
10	DEC Vax C and HP/Apollo C.
11	Metaware High C

If this option is not specified, then CodeCheck will assume that the source code is ANSI with common extensions (**–K3**).

If option **–K** is specified with no digit following, then CodeCheck will assume that the user meant strict K&R C (**–K0**).

–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 may follow immediately. If no name is given then the listing file will be 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 listed first as it is found in the source file, and second as it appears with all macros expanded. The **–L** option is assumed if **–M** is found. If **–L** is found without **–M**, then the listing file created by CodeCheck will not exhibit macro expansions.

–N Allow nested /* ... */ comments.

–NEST Allow C++ nested class definitions.

–O Append all CodeCheck stderr output to the file stderr.out. This is useful for those using the MS-DOS operating system, which does not permit the redirection 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, and each function definition as it is parsed.

–Q Specify a target directory. The pathname of the directory into which all CodeCheck output files are to be placed must follow immediately, *e.g.*

```
check -L -Q./temp mysource.c
```

Examples of such output files are the listing and prototype files. If this option is omitted CodeCheck will write its output files to the current working directory.

–R 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 mysource.c
```

CodeCheck first looks for a object (*i.e.* compiled) rule file of this name (*e.g.* foobar.cco). If this file is out-of-date or not found, CodeCheck will recom-

pile the rule file (foobar.cc) into an object file (foobar.cco) before proceeding to apply these rules to the source file.

More than one **–R** file may be specified: in this case all the rules will be compiled together into an object file named temp.cco.

If no **–R** file is specified, CodeCheck first looks for an object file named default.cco. If this file is found then it's rules are used. If it is not found then checking proceeds with no user-defined rules.

–S*n* Apply rules while reading header files. A digit should follow immediately, which identifies the kinds of header files:

0 No header files (***default***).
1 Headers enclosed in double quotes.
2 Headers enclosed in angle brackets.
3 All header files.

For example, suppose that these two lines are in a source file:

```
#include <ctypes.h>       //  A standard system header
#include "project.h"      //  An application header
```

When option **–S1** is in effect, CodeCheck will apply it's rules to *project.h* but not ctypes.h. *Please note that CodeCheck must* ***always*** *read every header included in a source file — this option only determines whether or not CodeCheck rules will be applied to the contents of the various headers.*

CodeCheck's default behavior is ***not*** to apply its rules to the contents of ***any*** included header files.

The environmental variable CCEXCLUDE, if it is used, takes precedence over this option. Rules are never applied to files that are found in directories listed in this variable.

–SQL Enables embedded SQL code. *Note:* this option must be spelled in all uppercase.

–T Create a file of prototypes for all functions defined in a project. The name of the prototype file may follow immediately. If no name is given then the name for the prototype file will be myprotos.h. The prototye 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 foo.c has the effect of treating foo.c as though it began with the preprocessor directive #undef MSDOS.

–V For CodeCheck users. See Section 1.4 for usage suggestions.

–W For CodeCheck users. See Section 1.4 for usage suggestions.

–X For CodeCheck users. See Section 1.4 for usage suggestions.

–Y For CodeCheck users. See Section 1.4 for usage suggestions.

–Z Suppress cross-module checking. Macro definitions and variable and function declarations will not be checked for consistency across the modules of a project. Often required in .CCP files when checking several source files at once.

Any letter of the alphabet may be used as a command-line option. Every option is remembered by CodeCheck and passed to the rule interpreter. CodeCheck rules can refer to ***and change*** these options by calling the functions option, set_option, str_option, and set_str_option (see Section 1.3–1.5 for details). Option **–X** is recommended for users who wish to design custom rule files whose behavior is controlled by a command-line option.

Batch processing [@file] of large command strings is done with CodeCheck .ccp files. See below CodeCheck File Names.

1.2 CodeCheck File Names

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 ccp.

.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. Any CodeCheck options may be listed in a ccp file, so long as each option is delimited by CR-LF. There is no limit to the number of ccp files on the command line.

Depending on command line options, the following 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 name of the object file created by CodeCheck when more than one rule file is compiled (several **–R** options).

1.3 Rules can Depend on Options

The CodeCheck function option() allows rules to behave differently depending on the options chosen by the user. For example, by testing the value of option('L') the rule can distinguish between users who have asked for a listing file and those who have not. Here is an example which issues different warnings depending on whether the user has requested a list file.

```
1    if ( dcl_any_upper )
2       {
3       if ( option('L') )   // This is for the list file:
4          warn(99, “Spell this name in lower case!”);
5       else                 // This is for the console:
6          warn(99, “Identifier %s should be lower case.”, dcl_name());
7       }
```

The message for the list file (“Spell this name…”) is appropriate because it will appear in context, directly below the offending line, with a marker under the identifier in question. The other message is more appropriate for the console, because it will be seen out of context.

Sometimes it is desirable for a CodeCheck rule actually *to change* one of its given options. The following rule, for example, will allow CodeCheck to decide that nested comments are okay as soon as it finds a nested /*.

```
1    if ( lin_nested_comment )
2       {
3       if ( ! option('N') )
4          set_option( 'N', 1 );
5       warn( 1234, “Nested comment.”);
6       }
```

1.4 Options can Transmit Parameters

The CodeCheck functions option() and str_option() allow the user to transmit numeric and string information to CodeCheck rules. All CodeCheck command-line options can be determined within CodeCheck rules. For example, if the user invokes CodeCheck with the command line:

check -V2 –Rerror test.c

then the function call str_option('R') will return the string "error", and option('V') will return the value 2. The former could be used to print messages that refer to the name of the rule file, and the latter could be used in a CodeCheck rule to define a “verbosity” level, for example:

```
1    if ( stm_depth > 6 )
2       switch( option('V') )
3          {
4          case 0:  /*  one-line message   */
5             break;
6          case 1:  /*  two-line message   */
7             break:
8          case 2:  /*  extended message   */
9             break;
10         }
```

The command-line options **–V**, **–W**, **–X**, **–Y** are guaranteed always to be available to users for any purpose. All other options have meanings pre-assigned by Abraxas Software, or are reserved for future use.

1.5 Rules can Set Default Options

Command-line options do not have to be specified in the command line itself. For example, the following rule sets up one programmer's normal options, so that he does not need to type them in his command line:

```
1    if ( prj_begin )
2       {
3       set_option('M',1);     // Expand macros in listing file.
4       set_option('B',1);     // Braces are part of compound statements.
5       set_option('E',1);     // Count macro-derived tokens too.
6
7       set_str_option( 'I', "C:\C600\INCLUDE" );
8       set_str_option( 'I', "C:\RUN286\INCLUDE" );
9       }
```

Defaults cannot be set on options **–K** and **–R**.

Chapter 2: CodeCheck Programming Concepts

2.1 CodeCheck Rules

A CodeCheck rule is a C if-statement, written using a restricted subset of the C grammar. The logical expressions that compose a rule refer to variables that are either declared in the CodeCheck program or are predefined by CodeCheck. Here is an example of two simple CodeCheck rules:

```
1    if ( pp_white_before > 0 )
2       warn( 2090, "Space before # is not portable.");
3
4    if ( pp_trailer )
5       warn( 2091, "Trailing tokens are not portable." );
```

The first rule uses the predefined variable pp_white_before, which becomes non-zero whenever a # character is found that is separated from a newline character by whitespace (*i.e.* space or tab characters). The CodeCheck function warn() echoes its arguments (an error number and a string) to the stderr stream, with an indication of the filename and line number at which the error was found. The warnings look like this:

```
test.c(124): warning W2090: Space before # is not portable.
test.c(126): warning W2090: Space before # is not portable.
test.c(127): warning W2091: Trailing tokens are not portable.
```

If a listing file is being made (option **–L**), then CodeCheck will also insert the warning message into the listing file after the offending line, with a letter (A, B, C, …) under the position of the error. The first error message for the line refers to the position marked with the letter A; the second is marked with the letter B; *etc.* The listing file will look like this:

```
   123     #ifdef BSD
   124        #include <sys/dir.h>
------>    A
A: warning W2090: Space before # is not portable.

   125     #else
     -        #include <dirent.h>
------>    A
A: warning W2090: Space before # is not portable.
```

```
   127     #endif BSD
------>        A
A: warning W2091: Trailing tokens are not portable.
```

Note that line 126 has no line number: this indicates that it was suppressed.

2.2 Rule Evaluation

The CodeCheck interpreter will evaluate a rule as often as necessary to assure its correct operation. Thus, rules which refer to low-level lexical variables will be evaluated most often during the code-checking process, while high-level rules will be evaluated least often. The order in which CodeCheck rules are found in a source rule file does not affect the order in which they are interpreted by CodeCheck.

A CodeCheck rule is triggered whenever its "if" condition is satisfied. Since CodeCheck rules can perform arithmetic and assign values to variables, it is quite possible for a CodeCheck rule to trigger other CodeCheck rules. This triggering happens immediately: as soon as the value of a user-declared CodeCheck variable changes, all other rules using this variable are triggered. Thus CodeCheck operates like a forward-chaining expert system, even though its rules are written in a procedural language.

It may be instructive to review the various ways in which CodeCheck both resembles and differs from a true expert system. For a program to be an expert system in the strict sense of the term, it must, as a minimum, have these three features:

1. A set of "rules", external to the program itself, expressed in either of these two forms:

   1a. if ***circumstances*** then ***actions***

   1b. if ***circumstances*** then ***conclusions***

2. A set of "facts" representing the current state of knowledge of the system.

3. A rule interpreter with the ability to use rules in more than one way. The three most common uses for rules are the following:

   3a. To repeatedly recognize ***circumstances*** and perform ***actions*** or assert ***conclusions*** until nothing further can be done. (This kind of logical inference is known as "forward chaining").

   3b. To verify ***conclusions*** by recursively testing ***circumstances***. (This is known as "backward chaining").

3c. To explain ***actions*** or ***conclusions*** by reference to the applicable chain of rules.

CodeCheck satisfies conditions (1) and (2), in that it has an external set of rules and facts of the required form. (CodeCheck "facts" are the values of its user- and pre-defined variables.) It also has a rule interpreter which recognizes circumstances and performs actions. However, CodeCheck uses rules in only one way — forward chaining.

Much of the power of expert systems derives from their flexible use of external bodies of facts and rules. These so-called "rule bases" encode the knowledge used by the expert system in a form that is (in the ideal case) understandable and maintainable by non-programmers. CodeCheck makes use of this significant source of structural power, but in a purely procedural way. It is thus a hybrid between a procedural interpreter and a logical expert system.

2.3 Rule Syntax

The syntax of a CodeCheck rule is almost the same as the syntax of an *if*-statement in C:

```
1    if ( expression )
2       statement
```

The ***expression*** in the rule condition is called the "trigger" for the rule, because it defines the event in which the rule is to be evaluated. The ***statement*** in the rule may be a compound statement surrounded by braces, just as in C. The only difference between a C *if*-statement and a CodeCheck rule is this: a rule can have no *else* statement. The reason for this is easy to see — the *else* statement, if it existed, would have to be evaluated whenever the trigger is not being triggered, an ill-defined event.

However, the ***statement*** in a rule may certainly contain *if*-statements, and these may have *else* statements. This syntax has the following format:

```
1    if ( trigger )
2       {
3       if ( expression )
4          statement1
5       else
6          statement2
7       }
```

There is no ill-defined event in this context, because the event causing the evaluation of the rule has been unambiguously defined by the trigger.

There are no local variables in CodeCheck — all variables are global, no matter where they are declared. Every user-defined CodeCheck variable must be declared before it is used.

The only kinds of control-flow statement permitted inside a CodeCheck rule are if, while, and switch statements. Break and continue are permitted.

The following are not permitted:

for do goto return

2.4 CodeCheck Operators

The C operators that are valid in CodeCheck expressions are the following:

( )	function call
[ ]	subscript selection
++	pre- and post-increment
--	pre- and post-decrement
+	unary and binary plus
-	unary and binary minus
*	multiply
/	divide
%	remainder
<<	left shift
>>	right shift
=	assign
+=	add and assign
-=	subtract and assign
*=	multiply and assign
/=	divide and assign
%=	remainder and assign
!	logical negation
==	logical comparison (equality)
<	logical comparison (less)
>	logical comparison (greater)

<=	logical comparison (=)
>=	logical comparison (=)
!=	logical comparison (?)
&&	logical conjunction
||	logical disjunction
~	bitwise complement
&	bitwise **AND**
|	bitwise **OR**
&	address-of operator

*The C operators that are **NOT VALID** in CodeCheck expressions are:*

.	struct and union member selection
->	pointer dereference
*	indirection operator
sizeof	size operator
<<=	left shift and assign
>>=	right shift and assign
^	bitwise **XOR**
&=	bitwise **AND** and assign
|=	bitwise **OR** and assign
^=	bitwise **XOR** and assign
? :	conditional operator
,	comma operator
::	C++ scope operator
.*	C++ object member selector
–>*	C++ object member pointer

2.5 CodeCheck Programs

A CodeCheck program is a collection of CodeCheck rules, optionally including declarations for user-defined variables. The C preprocessor is available within CodeCheck programs, and normal C comments can be placed anywhere. C++ single-line comments (delimited on the left by //) are also supported.

Here, in its entirety, is an example of a CodeCheck program that calculates the density of operators per line of C code. This rule set makes use of four predefined CodeCheck variables: fcn_begin and fcn_end, which flag the beginning and end of C functions, lin_operators, which counts the number of operators in each line of code, and fcn_total_lines, which counts the number of lines in each C function.

```
 1  float    ops,     /* number of operators    */
 2           density; /* operators per line     */
 3
 4  if ( fcn_begin )
 5     ops = 0.0;
 6
 7  if ( lin_end )
 8     ops += lin_operators;
 9
10  if ( fcn_end )
11     {
12     density = ops / fcn_total_lines;
13     printf( "Function %s:\n", fcn_name() );
14     printf( "   operator density = %g\n", density );
15     }
```

As CodeCheck scans a C source file, it interprets these rules in the following order. Every time a function definition is found, the rule on lines 4-5 is executed. Every time an end-of-line is found, the rule on lines 7-8 is executed. And lastly, every time the end of a function is found, the rule on lines 10-15 is executed. By this mechanism the variable *ops* accumulates the operator count until the end of the function, and is reset to zero at the start of the next function.

2.6 Predefined and User-defined Variables

There are well over 400 predefined CodeCheck variables which flag the occurrence of stylistic features and potential portability or maintenance problems. These variables describe features that range from the lowest level of lexical analysis all the way up to features of the project as a whole. A detailed description of each predefined CodeCheck variable may be found in Chapter 6. In addition to the predefined variables, the user can declare and use both integer and floating-point variables within any rule set.

2.6.1 All CodeCheck Variables are Global

Unlike C automatic and static variables, no CodeCheck variable is defined locally. All CodeCheck declarations are treated as though they were of storage class *extern*, *i.e.* global. At the beginning of every CodeCheck rule file there must be a declaration for every user-defined variable that is referred to in the file. Unlike C, you may not declare local variables within compound statements in CodeCheck.

2.6.2 Only Simple Types are Allowed

The only base types allowed for user-defined variables in this version of CodeCheck are int, float, and char. Structs, unions, arrays, pointers, and functions are not allowed. Strings (*i.e.* zero-terminated character arrays) are allowed only as the return values of CodeCheck functions, and as string literals (*e.g.* "this is a string literal"). A CodeCheck declaration may include an initializer, exactly as in C, but the initializer must be a constant, not an expression. Variables without explicit initializers are initialized to zero.

On 80x86 and 680x0 platforms, the size of a CodeCheck int is 32 bits, as is the size of a CodeCheck float. For other platforms consult the README file. The CodeCheck char type is signed.

Note: Currently, one-dimensional char arrays are allowed. When such kind of variable is declared, the dimension of the array must be specified. A pointer to char type is also allowed. Therefore a pointer of such kind can be used as index to a string.

2.6.3 CodeCheck Variables are Frequently Reset to Zero

It is very important to understand when CodeCheck predefined variables are re-reset to 0. Each predefined variable is set to 0 at the start of execution, and then again at the end of the scan of every grammatical object to which it refers. Consider, for example, the variable dcl_union_init, which is given the value 1 whenever a union initializer is found. This variable refers to declarations (as indicated by its prefix dcl_), and is therefore reset to 0 at the end of every declaration. Thus, its value is 0 until an initializer for a union is found, whereupon it is set to 1. It retains this value until the end of the declaration, at which time it is reset to 0.

The act of re-initializing a predefined CodeCheck variable does not cause any rules to be triggered. The CodeCheck interpreter refers to its rules only when the value of a CodeCheck variable is changed as a result of an explicit event. In the case of *predefined* variables, this event is the one described in the definition of the variable (definitions for all predefined variables are given in Chapter 6). In the case of a *user-defined* variable, the event is any circumstance in which its value is changed as a result of the interpretation of a rule.

2.6.4 CodeCheck has a Storage Class for Statistical Variables

The special statistic storage class is defined in CodeCheck for variables that are used for measurement purposes. This storage class is designed to simplify and optimize the calculation of statistical values (*e.g.* means, medians, quartiles, histograms, *etc.*) for software metrics. No other storage classes are permitted in CodeCheck.

A variable in the statistic storage class receives special treatment from CodeCheck. The major difference is that every value ever assigned to the statistic is remembered, so that statistical functions of these variables (*e.g.* mean, correlation) can be readily computed. Values of statistical variables are stored in the heap, and are freed with the CodeCheck reset() function.

Some CodeCheck predefined variables are of type statistic int. As a general rule, these are the CodeCheck variables that count features of functions and modules. For example, at the end of every function definition scanned by CodeCheck, the predefined variable fcn_operators is set to the number of standard C operators found in the line of code before macro expansion. Variables of storage class statistic have these properties:

1. Every value assigned to a statistical variable is treated as a separate observation or case. CodeCheck stores every case of every statistical variable that is used in a rule file.

2. The “rvalue” of a statistical variable is its most-recently assigned value. (An rvalue of a variable is the value used when the variable appears on the right-hand side of an assignment.)

3. CodeCheck statistical functions can be applied to statistical variables, *e.g.* mean(), median(), and histogram().

4. A statistical variable can be reset (all of its cases erased) with the CodeCheck reset() function.

5. The increment and decrement operators (++ and --) may not be used on statistical variables.

6. User-defined statistical variables may be statistic float or statistic int. They cannot be any other type.

Chapter 3: CodeCheck C/C++ Analysis Variables

3.1 Conversion Variables

All predefined CodeCheck variables that have the prefix cnv_ refer to characteristics of the implicit type conversions that frequently occur as executable operators are evaluated. Every CodeCheck conversion variable is initialized to zero at the start of execution, and again at the end of the scan of the operands of every executable operator.

cnv_any_to_bitfield	Set to 1 when an expression requires an implicit conversion from any type to a bitfield.
cnv_any_to_ptr	Set to 1 when an expression requires an implicit conversion from any non-pointer type to a pointer type.
cnv_bitfield_to_any	Set to 1 when an expression requires an implicit conversion from a bitfield to any type.
cnv_const_to_any	Set to 1 when an expression requires an implicit conversion from a constant type to any non-constant type.
cnv_float_to_int	Set to 1 when an expression requires an implicit conversion from a floating-point type to an integer type.
cnv_int_to_float	Set to 1 when an expression requires an implicit conversion from an integer type to a floating-point type.
cnv_ptr_to_any	Set to 1 when an expression requires an implicit conversion from a pointer type to any non-pointer type.

cnv_ptr_to_ptr	Set to 1 when an expression requires an implicit conversion from a pointer type to a different pointer type.
cnv_signed_to_any	Set to 1 when an expression requires an implicit conversion from a signed type to any unsigned type.
cnv_truncate	Set to 1 when an expression requires an implicit conversion from a larger arithmetic type to a smaller arithmetic type.

3.2 Declaration Variables

All predefined CodeCheck variables that have the prefix dcl_ refer to characteristics of declarators. Every CodeCheck declaration variable is initialized to zero at the start of execution, ***and again at the end of the scan of every declarator.*** If the declarator has an initializer, then reinitialization occurs at the end of the scan of the initializer.

A declarator declares the name, type, and initial value of a single variable or function (there can be more than one declarator in a declaration). For example, the following declaration has four declarators, of which two are functions and one has an initializer:

```
long        eeny, meeny = 1, miny(), moe();
```

The end of a declarator is marked by a comma or semicolon. CodeCheck evaluates declarators recursively, so that variables that refer to declarators that contain declarators are correctly set.

dcl_3dots	Set to 1 whenever an ellipsis (...) is found.
dcl_abstract	Set to 1 when an abstract declarator is found (*e.g.* the type name in a cast operator).
dcl_access	Set to 1 for C++ protected members, and set to 2 for C++ private members. (Note: this variable remains zero for public members, and therefore cannot act as a trigger for these members.)
dcl_aggr	Set to 1 whenever an array, union, struct, or class is declared.
dcl_all_upper	Set to 1 if only uppercase letters are found in an identifier name when it is declared.
dcl_ambig	If the current declarator name matches the name of another visible identifier on the first 6 or more characters, then this variable is set to the number of matching characters (see also dcl_extern_ambig).
dcl_any_upper	Set to 1 if an uppercase letter is found anywhere in an identifier name when it is declared.

dcl_array_size Set to the number of elements in an array whenever an array declarator is found. If no size is given, then this variable is set to –1. If the array is multidimensional, then this variable is set to the total size.

dcl_auto_init Set to 1 if an initializer for an *automatic* array, *struct*, or *union* is found.

dcl_base Set to an integer which identifies the base type of the current declarator. The base types are defined as manifest constants in the CodeCheck standard header file check.cch. These constants are:

```
#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
#define USHORT_TYPE            11
#define UINT_TYPE              12
#define ULONG_TYPE             13
#define EXTRA_UINT_TYPE        14 (*)
#define FLOAT_TYPE             15
#define SHORT_DOUBLE_TYPE      16 (***)
#define DOUBLE_TYPE            17
#define LONG_DOUBLE_TYPE       18
#define INT8_TYPE              19 (****)
#define INT16_TYPE             20 (****)
```

```
#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 (*)
#define CONSTRUCTOR_TYPE        30
#define DESTRUCTOR_TYPE         31
```

(*) **Note 1**: These are user-definable extensions to the C/C++ set of fundamental types. Use the function new_type() to introduce these types to CodeCheck.

(**) **Note 2**: DEFINED_TYPE is used when the base type is a typedef name. Use the CodeCheck function dcl_name() to obtain its name.

(***) **Note 3**: The unusual base types “long long” and “short double” are recognized as distinct types by CodeCheck. The type “long float” is considered to be equivalent to “double”.

(****) Note 4: Types `__int8, __int16, __int32 and __int64` are extended integral types in Microsoft Visual C++, Borland C++ etc.

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. For values. See `check.cch`.

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 — Set to 1 if a bitfield is found.

dcl_bitfield_anon	Set to 1 if an unnamed bitfield is found.
dcl_bitfield_arith	Set to 1 if a bitfield width requires calculation.
dcl_bitfield_size	Set to number of bits in a bitfield.
dcl_conflict	Set to 1 when an identifier was declared differently elsewhere. Use conflict_file() and conflict_line for location.
dcl_count	Set to the index of the current declarator within a comma-delimited declaration list. The first declarator has index 1, the second 2, *etc*, until the semicolon is found that marks the end of the list.
dcl_cv_modifier	Set to 1 if the keyword const is used as a non-ANSI type modifier (similar to near, far, *etc.*) rather than as an ANSI type specifier. Set to 2 if the keyword volatile is used as a non-ANSI type modifier.
dcl_definition	Set to 1 if a declaration is a definition, not a reference. Defining declarations reserve memory space, referencing declarations do not.
dcl_empty	Set to 1 if an empty declaration is found.
dcl_enum	Set to 1 when an enumerated constant is defined.
dcl_enum_hidden	Set to 1 when a declarator name hides an enumerated constant.
dcl_exception	C++ exception declaration complete.
dcl_explicit	Set to1 when a declarator has specifier "explicit".
dcl_extern	Set to 1 if the extern storage class is explicitly used in a declaration.
dcl_extern_ambig	If two external identifiers have names that agree on the first 6 or more characters, *regardless of case*, then this variable is set to the number of consecutive characters on which they agree.
dcl_first_upper	Set to the number of initial uppercase letters in an identifier when it is declared.
dcl_friend	Set to 1 if this is a friend declaration.

dcl_from_macro Set to 1 if the declarator name is derived from the expansion of a macro.

dcl_function Set to 1 if this is a function declaration.

dcl_function_flags If the current declarator is a function or a pointer to a function, then this variable is set to an integer which identifies the special characteristics of the function. These function characteristics are defined as manifest constants in the CodeCheck header file check.cch. These constants are:

```
#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
```

dcl_function_ptr Set to 1 if this declaration is a pointer to a function .

dcl_global Set to 1 if an identifier with external linkage has been declared. This includes variable, function, and typedef names.

dcl_hidden Set to 1 if an inner-block declaration hides an outer.

dcl_Hungarian Set to 1 if the Hungarian style is detected (a capital letter is immediately preceded by a lowercase letter).

dcl_ident_length Set to the number of characters in the declared identifier.

dcl_init_arith Set to 1 when a computed initializer is found, or when a computed explicit value for an enumerated constant is found.

dcl_initializer Set to 1 when an initializer is found.

dcl_inline Set to 1 when an inline function is declared.

dcl_label_overload Set to 1 if an inner-block declarator name matches a label within the same function.

dcl_levels	Set to the number of levels in the current declarator. Each of these counts as a level: *pointer to…*, *array of…*, *function returning…*, or (C++ only) *reference to…* For simple variables the value of dcl_levels is zero.
dcl_local	Set to 1 if a local identifier has been declared. This includes local variables, function parameters, typedef names, tag names, and enumerated constants. It does not include labels.
dcl_local_dup	Signal if a symbol is used more than once at current local scope. The Gnu-Compiler allows this declaration, but warns "shadowed variable". The value returned by dcl_local_dup was the line number last of the previous declaration.
dcl_long_float	Set to 1 if a variable or function is declared long float.
dcl_member	Set respectively to 1, 2, or 3 when a member of a union, struct, or class is declared. In C++ this includes data members, functions, typedef names, nested tag names, and enumerated constants.
dcl_mutable	*Set to 1 when an indentifier is declared 'mutable'.*
dcl_need_3dots	Set to 1 when an ellipsis (...) is needed in a function parameter list, but is not found.
dcl_new_array	Set to 1 when a nonstandard C++ array allocator "operator new[]" is declared.
dcl_no_specifier	Set to 1 if a variable or function declarator has no explicit type information. This variable is *not* triggered by C++ constructors and destructors.
dcl_no_prototype	Set to 1 if a function definition is found with no prior function prototype in scope. This variable is set even if this function definition is in prototype form.
dcl_not_declared	Set to 1 if an old-style function parameter is *not* declared. That is, the parameter is listed in the parameter list, but not declared in a formal declaration.
dcl_oldstyle	Set to 1 if an old-style (*i.e.* not prototyped) function is declared.

dcl_parameter Set to the index of a function parameter when one is found (1 for the first parameter, 2 for the second, *etc.*).

dcl_parm_count Set to the number of formal parameters found in a function declaration.

dcl_parm_hidden Set to 1 if a function parameter has the same name as an identifier declared within the function's compound statement.

dcl_pure Set to 1 if a pure member function is found.

dcl_simple Set to 1 when a simple variable (*i.e.* neither pointer, array, reference, nor function) is declared.

dcl_signed Set to 1 if the signed type specifier is explicitly used in a declaration.

dcl_static Set to 1 when a non-local static identifier has been declared.

dcl_storage_first Set to 1 when a storage class specifier is *preceded* by a type specifier (*e.g.* short static xyz).

dcl_storage_flags Set to an integer which identifies all of the storage class specifiers for the current declaration list. The specifier flags are defined as manifest constants in the CodeCheck header file check.cch. These constants are:

```
#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  (*)
```

(*) **Note**: Found only in the Vax C dialect.

dcl_tag_def Set to 1 when a new tag (enum, union, struct, or class) is defined as part of a type specifier. Set to 2 if the tag is anonymous (*i.e.* has no tag name).

dcl_template Set to the number of C++ template parameters if this is a function template.

dcl_throw_parameter A C++ throw argument. Created to support java-style exception checking.

dcl_type_before Set to 1 when the return type of a function definition is on the line *before* the line with the name of the function.

dcl_typedef Set to 1 if a typedef name has been declared. The name itself can be obtained by calling the CodeCheck function dcl_name().

dcl_typedef_dup Set to 1 whenever a duplicate type definition is found.

dcl_underscore Set to the number of leading underscores in the declarator name.

dcl_union_bits Set to 1 if a bitfield is declared as a member of a union.

dcl_union_init Set to 1 when a union initializer is found.

dcl_unsigned Set to 1 when the type specifier unsigned is used in a declaration.

dcl_variable Set to 1 if a variable is declared.

dcl_virtual Set to 1 if a virtual member function is declared.

dcl_zero_array Set to 1 whenever an array declarator is found that specifies a dimension of zero.

Associated CodeCheck variables:

conflict_line When dcl_conflict is triggered (when a declaration conflicts with an earlier declaration), this variable is set to the line number for the earlier declaration. The file name is returned by the CodeCheck function conflict_file().

Associated CodeCheck functions:

char * conflict_file(void)

When dcl_conflict is triggered (when a declaration conflicts with an earlier declaration), this function returns the name of the file for the earlier declaration. The line number is given by the variable. (See also conflict_line.)

char * dcl_array_dim(int k)

If the k^{th} level of type for this declarator is an array, then this function returns the array dimension (or –1 if no dimension was given).

char * dcl_base_name(void)

This function returns the name of the base type of the current declarator. If the base type is a typedef name then the typedef name is returned. If the base type is an enum, union, struct, or class, then the tag name is returned.

int dcl_level(int level)

Set to an integer which identifies the kind of the specified level (*function returning…*, *reference to…*, *pointer to…*, or *array of…*) for the current declarator. The number of levels for the current declarator is given by dcl_levels, which is zero for simple variables. The kinds are defined as manifest constants in the CodeCheck header file check.cch. These constants are:

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

int dcl_level_flags(int level)

Set to an integer which identifies all of the type qualifiers (*e.g.* const) of the specified level (*pointer to…*, *array of…*, *function returning…*, or *reference to…*) of the current declarator. The number of levels for the current declarator is given by dcl_levels, which is zero for simple variables. The last level always refers to the base type of the declarator. The level flags are defined as manifest constants in the CodeCheck header file check.cch. These constants are:

```
#define     CONST_FLAG      1       // ANSI
#define     VOLATILE_FLAG   2       // ANSI
#define     NEAR_FLAG       4       // DOS
#define     FAR_FLAG        8       // DOS
#define     HUGE_FLAG       16      // DOS
#define     EXPORT_FLAG     32      // Windows
#define     BASED_FLAG      64      // Microsoft
#define     SEGMENT_FLAG    128     // Microsoft
```

Here is an example of how dcl_level() and dcl_level_flags() work: suppose that a function *foo* is declared as

```
float far * const near * foo( int ) const;
```

In plain English, *foo* is a constant function of an integer returning a near pointer to a constant far pointer to a float. For the declarator *foo* dcl_levels will be set to 3. These levels are:

Level 0:	*constant function returning…*
Level 1:	*near pointer to…*
Level 2:	*constant far pointer to…*
Level 3:	*float*.

The flags for each level are returned by dcl_level_flags(k), where k runs through 0 … dcl_levels. In this example the values returned by this function are:

Level 0:	CONST_FLAG
Level 1:	NEAR_FLAG
Level 2:	CONST_FLAG + FAR_FLAG
Level 3:	0

char * dcl_name(void)

If CodeCheck is scanning a declarator, then this function returns the name of the current declarator, otherwise 0.

char * dcl_scope_name(void)

This function returns the class scope name right before the declarator. If the declarator is not scoped, the function returns 0.

int prefix(char * str)

This function returns 1 if the current declarator name or tag name begins the letters in str, otherwise 0. Each subsequent call to prefix **within the same rule** will start looking for the specified string at the character position immediately following the last successfully recognized prefix. Thus prefix can be used to parse sequences of prefixes.

char * root(void)

Returns the root of an identifier after application of functions prefix and/or suffix. For example, after calling prefix("foo_") on the identifier foo_bar, the function root() will return the string "bar".

int suffix(char * str)

This function returns 1 if the current declarator name or tag name ends the letters in str, otherwise 0. Each subsequent call to suffix **within the same rule** will start looking for the specified suffix at a position that precedes the last successfully recognized suffix. Thus suffix can be used to parse sequences of suffixes.

void new_type(char * name, int type)

This function informs CodeCheck of the existence of a nonstandard built-in keyword for a base type that is not defined in any header file. The first argument for new_type() should be the new keyword itself, in quotes. The second argument should be any of the possible values of dcl_base (which are defined as manifest constants in the standard CodeCheck header check.cch) *except* DEFINED_TYPE. If the value is one of the following:

```
#define EXTRA_INT_TYPE       6  //  e.g. Macintosh comp type
#define EXTRA_UINT_TYPE     11
#define EXTRA_FLOAT_TYPE    15  //  e.g. Macintosh extended type
#define EXTRA_PTR_TYPE      21  //  e.g. Microsoft _segment type
```

then CodeCheck will treat the new keyword as a new unique base type. If it is any other value then the keyword will be considered a synonym for the specified C type. Consult check.cch for the complete list of base types.

For example, let us suppose that a compiler has two base type keywords that are not part of standard C, namely long64, which stands for a 64-bit integer, and float80, which stands for an 80-bit floating-point type. This rule could be inserted into every CodeCheck rule file to handle these keywords:

```
1    if ( prj_begin )
2       {
3       new_type( "long64", EXTRA_INT_TYPE );
4       new_type( "float80", LONG_DOUBLE_TYPE );
5       }
```

In this example long64 has been introduced as an integer base type, not equivalent to any other integer base type, while float80 has been introduced as a simple synonym for the standard C base type long double.

On Macintosh systems, CodeCheck understands the base types extended and comp to correspond to LONG_DOUBLE_TYPE and EXTRA_INT_TYPE, respectively. These Macintosh keywords do not have to be defined by the Code-Check user.

On DOS systems, CodeCheck understands the Microsoft _segment keyword to refer to the EXTRA_PTR_TYPE base type. *Note:* the Microsoft _segment keyword is ***not*** the same as the Borland _seg keyword. The former is a base type, while the latter is a type modifier for pointers.

3.3 Expression Variables

All predefined CodeCheck variables that have the prefix exp_ refer to characteristics of C expressions. Unless otherwise noted, every expression variable is initialized to zero at the start of execution, ***and again at the end of the scan of every statement.*** CodeCheck evaluates expressions recursively, so that variables that refer to expressions that contain expressions are correctly set.

exp_empty_initializer	Set to 1 when an empty initializer is found.
exp_not_ansi	Set to 1 if a non-ANSI expression is found. This variable does *not* trigger on C++ expressions.
exp_operands	Set to the number of operands found in an expression, before macro expansion.
exp_operators	Set to the number of standard C operators found in an expression, before macro expansion.
exp_tokens	Set to the number of tokens found in an expression, before macro expansion.

Associated CodeCheck expression-functions:

exp_base_name()	Class base-name of current expression. Useful for obtaining resultant class-base-name of an overloaded function and/or pointer linked overloaded functions.

3.4 Function Variables

All predefined CodeCheck variables that have the prefix fcn_ refer to characteristics of C functions. Every function variable is initialized to zero at the start of execution, ***and again at the end of the scan of every function.*** The special variables fcn_begin and fcn_no_header are triggered when the beginning of a function definition is found.

fcn_aggr	Set to the number of array, union, struct, or class variables that are declared in a function (**statistic**).
fcn_array	Set to the number of local array elements declared in a function (**statistic**).
fcn_begin	Set to 1 when the beginning of a function has been found.
fcn_com_lines	Set to the number of pure comment lines in the definition of a C function (**statistic**).
fcn_decisions	Set to the number of binary decision points in a function (**statistic**).
fcn_end	Set to 1 when the end of a function has been found.
fcn_exec_lines	Set to the number of executable lines in the definition of a C function (**statistic**).
fcn_H_operands	Set to the total number of Halstead operands found in a function, before macro expansion (**statistic**).
fcn_H_operators	Set to the total number of Halstead operators found in a function before macro expansion (**statistic**).
fcn_high	Set to the number of high-level statements found in the definition of a C function (**statistic**).
fcn_locals	Set to the number of local variables declared in a function, including all nested compound statements (**statistic**).

fcn_low	Set to the number of low-level statements found in the definition of a C function (**statistic**).
fcn_members	Set to the number of local union, struct, or class members declared in a function (**statistic**).
fcn_no_header	Set to 1 when a function definition is found that is not preceded by a stand-alone comment block.
fcn_nonexec	Set to the number of non-executable statements found in the definition of a C function (**statistic**).
fcn_operands	Set to the total number of operands found in a function, before macro expansion (**statistic**).
fcn_operators	Set to the total number of standard C operators found in a function before macro expansion (**statistic**).
fcn_register	Set to the number of register variables declared within the current function.
fcn_simple	Set to the number of local simple variables (char, short, long, int, float, double) declared in a function (**statistic**).
fcn_tokens	Set to the total number of tokens found in a function before macro expansion (**statistic**).
fcn_total_lines	Set to the total number of lines in the definition of a C function (**statistic**).
fcn_u_operands	Set to the number of unique operands found in a function, before macro expansion (**statistic**).
fcn_u_operators	Set to the number of unique operators found in a function before macro expansion (**statistic**).
fcn_uH_operands	Set to the number of unique Halstead operands found in a function, before macro expansion (**statistic**).
fcn_uH_operators	Set to the number of unique Halstead operators found in a function before macro expansion (**statistic**).

fcn_unused Set to the number of local variables declared in a function but never used, including all nested compound statements (**statistic**).

fcn_white_lines Set to the number of whitespace lines in the definition of a C function (**statistic**).

Associated CodeCheck functions:

char * fcn_name(void)

This function returns the name of the function that is currently being checked.

Associated CodeCheck Variable:

stm_return_void Set to 1 if: (1) a return has no value in a function declared to return a non-void type, (2) if a function has no return statement but requires a returned value, or (3) if a return has a value in a function declared to return void.

3.5 Identifier Variables

All predefined CodeCheck variables that have the prefix idn_ refer to characteristics of variable and function names. Every CodeCheck identifier variable is initialized to zero at the start of execution, and again at the end of the scan of every variable and function used within executable code (*i.e.* not within declarations or preprocessor directives).

idn_base	Set to an integer which identifies the base type of this identifier, using the same values as dcl_base (section 3.2).
idn_bitfield	Set to 1 when an identifier is a named bitfield.
idn_constant	Set to 1 when an identifier is an enum constant.
idn_exception	Actual usage of C++ exception in code.
idn_exception_base	C++ exception base-type at trigger point idn_exception. See check.cch for values returned.
idn_function	Set to 1 when an identifier is a function name.
idn_global	Set to 1 when an identifier has file scope and external linkage.
idn_levels	Set to the number of levels in the type of the identifier. Each of these counts as a level: *pointer to…*, *array of…*, *function returning…*, and *reference to…* For simple variables the value of idn_levels is zero.
idn_line	Set to the line number of the declaration in scope for the identifier.
idn_local	Set to 1 when an identifier has local scope (*i.e.* it was declared within a function body).
idn_member	Set to 1 when a C++ identifier has class scope.
idn_no_init	Set to 1 if this identifier is a local variable whose value is being used before it has not been initialized.

idn_no_prototype	Set to 1 if a function call is found without a prototype for the function in scope.
idn_not_declared	Set to 1 when a function is called without any declaration in scope.
idn_parameter	Set to 1 when an identifier is a function parameter.
idn_storage_flags	Set to an integer which identifies the storage class of the identifier, using the same values as dcl_storage_flags (section 3.2).
idn_variable	Set to 1 when an identifier is a variable.

Associated CodeCheck functions:

char * idn_array_dim(int k)

If the k^{th} level of type for this identifier is an array, then this function returns the array dimension (or –1 if no dimension was given in the declaration).

char * idn_base_name(void)

If the base type of the identifier is a tag (enum, union, struct, or class) or typedef name, then this function returns the tag or typedef name as a character string.

char *idn_exception_name()

Name of exception currently being used at trigger point idn_exception.

char * idn_filename(void)

Returns the name of the file that contains the declaration in scope for the identifier. (See also idn_line).

int idn_level(int k)

This function returns the kind of the k^{th} level of the identifier, using the same values as dcl_level(k) (section 3.2).

int idn_level_flags(int k)

This function returns the flags for the k^{th} level of the identifier, using the same values as dcl_level_flags(k) (section 3.2).

char * idn_name(void)

Returns the name of the identifier.

3.6 Lexical Variables

All predefined CodeCheck variables that have the prefix *lex_* refer to characteristics of lexical tokens, *i.e.* the names, numbers, operators, and punctuation marks that comprise a C program. Every lexical variable is initialized to zero at the start of execution, ***and again at the end of the scan of every token.***

lex_ansi_escape — Set to 1 if an escape sequence contains one of the new ANSI escape characters: a, v, or ?.

lex_assembler — Set to 1 if embedded assembler code is detected.

lex_backslash — Set to 1 if a backslash-newline pair is found at the end of a line that is not part of a macro definition.

lex_bad_call — Set to the difference between the number of arguments found and the number of arguments expected when a macro function is expanded.

lex_big_octal — Set to 8 or 9, respectively, when a numeric escape sequence or octal integer contains the digits 8 or 9.

lex_c_comment — Set to 1 when a C comment (/* ... */) is found.

lex_cpp_comment — Set to 1 when a C++ comment (//...) is found.

lex_char_empty — Set to 1 if an empty character constant is found (*e.g.* ''). This variable does *not* flag the null character constant ('\0').

lex_char_long — Set to 1 if a character constant is longer than one character.

lex_enum_comma — Set to 1 when a list of enumerated constants ends with a comma.

lex_constant — When a constant is found this variable is set to one of the following values, defined as manifest constants in the CodeCheck standard header file check.cch.The values are:

```
#define     CONST_BOOL      1
#define     CONST_ENUM      2
#define     CONST_CHAR      3
```

```
#define     CONST_INTEGER     4
#define     CONST_FLOAT       5
#define     CONST_STRING      6
```

lex_float Set to 1 if a numeric constant is found with the suffix 'F' or 'f'.

lex_hex_escape When a hexadecimal escape sequence is found, this variable is set to number of hexadecimal digits found.

lex_initializer When an initializer is found this variable is set to one of the following values, defined as manifest constants in the CodeCheck header file check.cch.The values are:

```
#define     INIT_ZERO         1
#define     INIT_INTEGER      2
#define     INIT_BOOL         3
#define     INIT_CHAR         4
#define     INIT_FLOAT        5
#define     INIT_STRING       6
#define     INIT_OTHER        7
```

lex_intrinsic Set to 1 whenever an identifier is a function name that is intrinsic to (*i.e.* predefined by) the compiler chosen through the –K command-line option.

lex_invisible Set to 1 when an *unscoped* identifier is visible to ANSI C and all versions of C++ prior to 3.0, but is invisible to C++ 3.0.

lex_key_no_space Set to 1 when a keyword is not followed by a space.

lex_keyword Set to 1 when a keyword is found.

lex_lc_long Set to 1 if a constant is found with the suffix 'l' (lower case “el”).

lex_long_float Set to 1 if a floating constant is found with the suffix 'L' or 'l' (upper or lower case “el”).

lex_long_long Signal "long long" 64 bit type.

lex_macro Set to 1 whenever a macro is about to expand. Call function token() for the name of the macro to be expanded.

lex_macro_token	Set to 1 if a lexical token originates from a macro expansion.
lex_metaware	Set to 1 whenever a Metaware High C lexical extension to the C language is found.
lex_nl_eof	Set to 1 if a non empty source file does not end with a newline.
lex_nonstandard	Whenever a character is found that is not in the standard C set, the value of this variable is set to the integer representation of the nonstandard character.
lex_not_KR_escape	Whenever an escape character is found that is not defined by K&R (*i.e.* \n, \b, \t, \r, \f, \\, \", \') then this variable is set to the integer representation of the character.
lex_not_manifest	Set to 1 if a numeric constant other than 0 or 1 is used in any context other than a macro definition or a comment.
lex_null_arg	Set to 1 if an actual argument is omitted in a macro call, *e.g.* XYZ(abc,,123).
lex_num_escape	Whenever a non zero numeric escape sequence is found, the value of this variable is set to the value of the numeric escape sequence.
lex_punct_after	Set to 1 if a comma or semicolon is not followed by a whitespace character, a comma, or a semicolon.
lex_punct_before	Set to 1 if a comma or semicolon is preceded by a space.
lex_radix	Set to the radix of an integer constant (2, 8, 10, or 16).
lex_str_concat	Set to 1 if two string constants are found, separated only by whitespace.
lex_str_length	Set to the length of a string literal (the terminating zero is not counted).
lex_str_macro	Set to 1 when a macro identifier is found within a string constant.

lex_str_trigraph	Set to 1 if a trigraph is found in a string literal.
lex_suffix	Set to 1 if a numeric constant is found with any suffix (F, f, L, l, U, or u), or combination of these.
lex_token	Set to the index of the token within the current line (1 for first, 2 for second, *etc*.) whenever a token is found.
lex_trigraph	Set to 1 if an ANSI trigraph is found (anywhere).
lex_uc_long	Signal 'L' long type constant.
lex_unsigned	Set to 1 if a numeric constant is found with the suffix 'U' or 'u'.
lex_wide	Set to 1 if an ANSI wide string or character constant is found (prefix L).
lex_zero_escape	When a zero numeric escape sequence is found, this variable is set to 1 if the escape is within a character literal, or 2 if the escape is within a string literal.

Associated CodeCheck functions:

int identifier(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns the value 1 whenever an identifier (a variable or function name) has been encountered that matches the given string.

void ignore(char * name, ...)

This function causes the CodeCheck lexical analyzer to ignore any token that matches one of the argument names. Every argument must be a string.

int keyword(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns the value 1 whenever a keyword has been encountered that matches the given string.

char next_char(void)

This function returns the lexical analyzer's lookahead character: the character in the CodeCheck input stream that immediately follows the current token. This function may not be used as a rule trigger.

char * prev_token(void)

This function returns the previous token that has being parsed by CodeCheck. The token is in string form.

void skip_nonansi_ident(char c)

Skip non-ANSI identifiers beginning with '@','$' or '`'. The char parameter of this function specifies the character which leads the identifier. The value of the parameter only can be '@', '$' or '`'. The other characters have no effect for this function.

char * token(void)

This function returns the current token that has being parsed by CodeCheck. The token is in string form.

3.7 Line Variables

All predefined CodeCheck variables that have the prefix lin_ refer to characteristics of lines of C code. Every line variable is initialized to zero at the start of execution, ***and again at the end of the scan of every line.*** The end of a line is marked by a newline character, or by a backslash-newline pair.

Note that lines of C code are conceptually different from simple C statements, even though most programmers usually place no more than one statement on each line. It is nevertheless possible to have more than one statement on a line, or more than one line for a statement. For CodeCheck variables that apply to *statements*, see the next section of this chapter.

lin_continuation	Set to 1 if a line continues an expression or declaration list from the previous line.
lin_continues	Set to 1 if a line ends before the end of the current expression.
lin_dcl_count	Set to the number of identifiers declared on the current line (includes tag definitions and function parameters).
lin_depth	Set to the #include file nesting level.
lin_end	Set to 1 when an end-of-line marker has been found (a newline character or the backslash-newline pair).
lin_has_code	Set to 1 if a line contains code.
lin_has_comment	Set to 1 if a line has a comment that contains text.
lin_has_label	Set to 1 if a line contains a label.
lin_header	Set to 1 if the current line was obtained from a *project* header file via #include "filename". Set to 2 if the current line was obtained from a *system* header file via #include <filename>.
lin_include_kind	If the line is a preprocessor line with #include. Set to 1 if the file included is a user header file, *e.g.* “hdr.h”.

	Set to 2 if the file included is a system header files *e.g.* <hdr.h>
lin_indent_space	Set to the number of leading *space* characters found before the first non-white non-comment character of a line.
lin_indent_tab	Set to the number of leading *tab* characters found before the first non-white non-comment character of a line.
lin_is_comment	Set to 1 if a line has no C code and either contains a comment or is contained within a comment. The comment must contain text to qualify as a real comment.
lin_is_white	Set to 1 if a line consists entirely of whitespace (tabs & spaces), or is a comment line without any text.
lin_is_exec	Set to 1 if a line contains code that is executable.
lin_length	Set to the number of characters in the line (excluding the newline character at the end of the line).
lin_nest_level	Set to each line's nominal indentation level. This value may differ from the actual indentation of the line (for actual indentation, see *lin_indent_tab* and *lin_indent_space*). CodeCheck assumes that curly braces are to be indented only if the command line option **–B** has been given.
lin_nested_comment	Set to 1 if a /*...*/ comment is found nested within another /*...*/ comment.
lin_number	Set to the number of the current line, relative to the start of the current file.
lin_operands	Set to the number of operands found in a line of code, before macro expansion.
lin_operators	Set to the number of standard C operators found in a line of code before macro expansion.
lin_preprocessor	Set to a non-zero value representing the type of a preprocessor line (*i.e.* begins with #). The manifest

constants are defined in the CodeCheck header file check.cch. The constants are:

```
#define DEFINE_PP_LIN      1    /* #define */
#define UNDEF_PP_LIN       2    /* #undef */
#define INCLUDE_PP_LIN     3    /* #include */
#define IF_PP_LIN          4    /* #if */
#define IFDEF_PP_LIN       5    /* #ifdef */
#define IFNDEF_PP_LIN      6    /* #ifndef */
#define ELSE_PP_LIN        7    /* #else */
#define ELIF_PP_LIN        8    /* #elif */
#define ENDIF_PP_LIN       9    /* #endif */
#define PRAGMA_PP_LIN      10   /* #pragma */
#define LINE_PP_LIN        11   /* #line */
#define ERROR_PP_LIN       12   /* #error */
#define ASM_PP_LIN         13   /* #asm */
#define ENDASM_PP_LIN      14   /* #endasm */
#define C_INCLUDE_PP_LIN   15   /* #c_include */
#define R_INCLUDE_PP_LIN   16   /* #r_include */
#define RC_INCLUDE_PP_LIN  17   /* #rc_include */
#define INC_NEXT_PP_LIN    18   /* #include_next */
#define OPTION             19   /* #option */
```

lin_source — Set to 1 if the current line was *not* obtained from a header file.

lin_suppressed — Set to 1 if compilation of the current line has been suppressed by the preprocessor.

lin_tokens — Set to the number of tokens found in a line of code before macro expansion.

lin_within_class — Set to 1 if the current line is inside a C++ class definition, or 2 if it is within a member function definition that is outside the class definition.

lin_within_function — Set to 1 if the current line is inside a function definition.

lin_within_tag — Set to 1 if the current line is within an enumeration, 2 if it is within a union, 3 if it is within a struct, and 4 if it is within a class. This variable is *not* set for lines within class member function definitions.

Associate CodeCheck functions:

char * lin_include_name(void)

This function returns the file name included if this line is a preprocessor line with #include.

char * line(void)

This function returns the current input line as a null-delimited string. This string does not end with a new-line character.

3.8 Module Variables

All predefined CodeCheck variables that have the prefix mod_ refer to characteristics of modules, *i.e.* independently compilable source files with the .c filename extension, not header files with the .h extension. Every module variable is initialized to zero at the start of execution, ***and again at the end of the scan of every module.*** The special variable mod_begin is triggered just before a module is read.

Associated CodeCheck functions:

char * mod_name(void)

This function returns the name of the module that is currently being checked. A "module" is a C source file and all of its header files. Its name is the name of the first source file in the module.

int mod_class_lines(int index)

This function returns the total number of lines in each named class, struct, or union defined in the module, indexed by its order within the module. These lines *include* lines in definitions of member functions that are outside the class definition. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

char * mod_class_name(int index)

This function returns the name of each named class, struct, or union defined in the module, indexed by its order within the module. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

int mod_class_tokens(int index)

This function returns the number of tokens in each named class, struct, or union defined in the module, indexed by its order within the module. Tokens

in definitions of member functions defined outside the class definition are *included* in the token count. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

3.9 Operator Variables

All predefined CodeCheck variables that have the prefix op_ refer to characteristics of operators. Every operator variable is initialized to zero at the start of execution. When an operator is encountered the relevant operator variables are set. All operator variables are reset to zero immediately after all relevant rules have been triggered. There are three distinct kinds of operators handled here: executable operators (high-, medium-, and low-precedence), keyword operators (for executable keywords like if, while, and return), and non-executable operators (for punctuation marks and declarator symbols).

The operands of an operator are counted in the order from right to left. For example, to get an operand's information with a function such as op_base(), op_levels() etc. from expression “a + b”, the first operand is b and the second operand is a.

3.9.1 High Precedence Operators

op_address — Set to 1 when an address-of (&) operator is executed (one operand). **Note**: this is not the same as the “reference to” declarator symbol , which has the same appearance.

op_arrow — Set to 1 when an indirect member selector (->) is executed (one operand).

op_bit_not — Set to 1 when a bitwise NOT (~) operator is executed (one operand).

op_call — Set to 1 when a function is called (*i.e.* just after the close parenthesis that ends a function argument list is found in executable code). The number of arguments is given by op_operands.

op_call_overload — This C++ method call is overloaded. Function return type is context dependent. Function exp_base_name() may be used to determine actual return class base-name chosen.

op_catch — The "catch" keyword is set to one when the C++ error handling keyword is seen.

op_delete	Set to 1 when the C++ keyword delete is executed (one operand).
op_indirect	Set to 1 when an indirection (*) operator is executed (one operand). **Note**: this is not the same as the non-executable “pointer to” declarator symbol, which has the same appearance.
op_log_not	Set to 1 when a logical negation (!) operator is executed (one operand).
op_member	Set to 1 when a direct member selector (.) is executed (one operand).
op_memptr	Set to 1 when a C++ member pointer (–>*) is executed (two operands).
op_memsel	Set to 1 when a C++ member selector (**.***) is executed (two operands).
op_negate	Set to 1 when an arithmetic negation (-) operator is executed (one operand).
op_new	Set to 1 when the C++ keyword *new* is executed. The number of operands found is given by op_operands.
op_plus	Set to 1 when the unary plus (+) operator is executed (one operand). Note: this is ***not*** the binary add operator.
op_post_decr	Set to 1 when a postfix decrement (--) operator is executed (one operand).
op_post_incr	Set to 1 when a postfix increment (++) operator is executed (one operand).
op_pre_decr	Set to 1 when a prefix decrement (--) operator is executed (one operand).
op_pre_incr	Set to 1 when a prefix increment (++) operator is executed (one operand).
op_sizeof	Set to 1 when a *sizeof* operator is evaluated (one operand).
op_subscript	Set to 1 when a subscript ([]) is evaluated (two operands).

op_throw	The C++ "throw" keyword.
op_try	The C++ "try" keyword.

3.9.2 Medium Precedence Operators

op_add	Set to 1 when an add (+) operator is executed (two operands).
op_based	Set to 1 when a Microsoft “based” (:>) operator is executed (two operands).
op_bit_and	Set to 1 when a bitwise AND (&) operator is executed (two operands).
op_bit_or	Set to 1 when a bitwise OR (|) operator is executed (two operands).
op_bit_xor	Set to 1 when a bitwise XOR (^) operator is executed (two operands).
op_cast	Set to 1 when a cast is executed (two operands: the first is the operand to be type-cast, the second is the result type).
op_div	Set to 1 when a division (/) operator is executed (two operands).
op_equal	Set to 1 when a == operator is executed (two operands).
op_init	Set to 1 when the initialization operator is evaluated (one operand).
op_left_shift	Set to 1 when a left shift (<<) operator is executed (two operands).
op_less	Set to 1 when a less than (<) operator is executed (two operands).
op_less_eq	Set to 1 when a <= operator is executed (two operands).
op_log_and	Set to 1 when a logical conjunction (&&) operator is executed (two operands).

op_log_or	Set to 1 when a logical disjunction (||) operator is executed (two operands).
op_more	Set to 1 when a more than (>) operator is executed (two operands).
op_more_eq	Set to 1 when a >= operator is executed (two operands).
op_mul	Set to 1 when a multiplication (*) operator is executed (two operands).
op_not_eq	Set to 1 when a != operator is executed (two operands).
op_rem	Set to 1 when a remainder (%) operator is executed (two operands).
op_right_shift	Set to 1 when a right shift (>>) operator is executed (two operands).
op_subt	Set to 1 when a subtract (-) operator is executed (two operands).

3.9.3 Low Precedence Operators

op_add_assign	Set to 1 when a += operator is executed (two operands).
op_and_assign	Set to 1 when a &= operator is executed (two operands).
op_assign	Set to 1 when an assignment (=) operator is executed (two operands).
op_assoc	Set to 1 when the Metaware association operator (=>) is executed (two operands).
op_cond	Set to 1 when a conditional (?) operator is executed (three operands).
op_div_assign	Set to 1 when a /= operator is executed (two operands).
op_iterator_call	Set to 1 when a Metaware iterator operator (<–) is executed (two operands).

op_left_assign	Set to 1 when a <<= operator is executed (two operands).
op_mul_assign	Set to 1 when a *= operator is executed (two operands).
op_or_assign	Set to 1 when a |= operator is executed (two operands).
op_rem_assign	Set to 1 when a %= operator is executed (two operands).
op_right_assign	Set to 1 when a >>= operator is executed (two operands).
op_sub_assign	Set to 1 when a -= operator is executed (two operands).
op_xor_assign	Set to 1 when a ^= operator is executed (two operands).

3.9.4 Punctuation Marks and Declarator Symbols

op_close_angle	Set to 1 when the close angle bracket (>) of a C++ template argument list is found
op_close_brace	Set to 1 when a close curly brace (}) is found.
op_close_bracket	Set to 1 when a close bracket (]) is found.
op_close_funargs	Set to 1 when the close parenthesis of a function argument list is found.
op_close_paren	Set to 1 when a close parenthesis for a subexpression has been found.
op_colon_1	Set to 1 when a "unary" colon is found (*e.g.* in the label default:).
op_colon_2	Set to 1 when a "binary" colon is found (*e.g.* in the expression x = flag ? 0 : 1).
op_comma	Set to 1 when a *operator* (not the comma *separator*) is found. See op_separator.

op_destroy	Set to 1 when the C++ destructor symbol (~) is found.
op_iterator	Set to 1 when a Metaware iterator symbol (–>) is found.
op_macro_arg	A macro function call argument. Set to 1 when a macro argument is parsed. Fires between op_macro_begin and op_macro_call.
op_macro_begin	Set to 1 when a macro function call is first seen.
op_macro_call	Set to 1 when a macro function is about to be expanded, e.g. all arguments have been collected.
op_open_angle	Set to 1 when the open angle bracket (<) of a C++ template argument list is found.
op_open_brace	Set to 1 when an open curly brace ({) is found.
op_open_bracket	Set to 1 when an open bracket ([) is found.
op_open_funargs	Set to 1 when the left parenthesis of a function argument list is found.
op_open_paren	Set to 1 when an open parenthesis for a subexpression has been found.
op_pointer	Set to 1 when the “pointer-to” (*) declarator symbol is found. **Note**: this is not the same as the indirection operator (see op_indirect).
op_reference	Set to 1 when the “reference-to” (&) declarator symbol is found. **Note**: this is not the same as the address operator (see op_address).
op_scope	Set to 1 when a C++ scope (::) symbol is found.
op_semicolon	Set to 1 when a semicolon is found.
op_separator	Set to 1 when the comma expression separator is found. **Note**: this is not the same as the comma operator (see op_comma).

3.9.5 Halstead Keyword Operators

op_break	Set to 1 when the *break* keyword is found.
op_continue	Set to 1 when the *continue* keyword is found.
op_do	Set to 1 when the *do* keyword is found.
op_else	Set to 1 when the *else* keyword is found.
op_for	Set to 1 when the *for* keyword is found.
op_goto	Set to 1 when the *goto* keyword is found.
op_if	Set to 1 when the *if* keyword is found.
op_return	Set to 1 when the *return* keyword is found.
op_switch	Set to 1 when the *switch* keyword is found.
op_while_1	Set to 1 when the *while* keyword is found (unless it is part of a *do-while* construct).
op_while_2	Set to 1 when the *while* keyword is found as part of a *do-while* construct.

3.9.6 Operator Descriptors *(not alphabetical)*

op_operands	Set to the number of operands expected whenever an *executable* operator is found. If the operator is a function call (op_call), then this variable is set to the number of actual arguments in the argument list. If the operator is a cast (op_cast), then this variable is set to 2 (the first operand is the type of the operand to be type-cast, the second is the result type).
op_punct	Set to 1 for punctuation marks.
op_low	Set to 1 for operators with low precedence.
op_medium	Set to 1 for operators with medium precedence.
op_high	Set to 1 for high precedence operators.

op_keyword	Set to 1 if this is token is a keyword that would be considered by Halstead to be an operator: *if*, *else*, *do*, *for*, *while*, *switch*, *break*, *continue*, *goto*, and *return*.
op_Halstead	Set to 1 for those tokens that Halstead would have considered to be operators. These ***include*** all operators found within executable code and certain keywords (if, else, do, for, while, switch, break, continue, goto, and return), but ***exclude*** all operators found within declarations.
op_executable	Set to 1 if this is a standard operator that is executable.
op_declarator	Set to 1 if the operator is within a declaration, *not* including initializers.
op_bitwise	Set to 1 when any bitwise operator is found.
op_prefix	Set to 1 for unary prefix operators.
op_infix	Set to 1 for binary infix operators.
op_postfix	Set to 1 for unary postfix operators.
op_cast_to_ptr	Set to 1 when a cast operator has the form (*type* *).
op_space_before	Set to 1 if an operator or punctuation mark is preceded by a space character.
op_space_after	Set to 1 if an operator or punctuation mark is followed by a space character.
op_white_before	Set to 1 if an operator or punctuation mark is preceded by whitespace.
op_white_after	Set to 1 if an operator or punctuation mark is followed by whitespace.

Associated CodeCheck functions:

char * op_array_dim(int j, int k)

When an operator is executed, if the k^{th} level of the j^{th} operand is an array, then this function returns the array dimension (or –1 if no dimension was given).

int op_base(int j)

When an operator is executed, this function returns the base type of the j^{th} operand, using the same values as dcl_base (section 3.2).

char * op_base_name(int j)

If the base type of the j^{th} operand of an executable operator is a tag (enum, union, struct, or class) or typedef name, then this function returns the tag or typedef name as a character string.

int op_bitfield(int j)

Returns 1 if the base type of the j^{th} operand of an executable operator is a bitfield, otherwise zero.

char * op_function(void)

When op_open_funargs, op_close_funargs, or op_call is triggered, this function returns the name of the function that is to be declared or called. When an expression has to be evaluated in order to find which function to call, *e.g.* through a function pointer, then this function returns an empty string.

int op_level(int j, int k)

When an operator is executed, this function returns the kind of the k^{th} level of the j^{th} operand, using the same values as dcl_level(k) (section 3.2).

int op_level_flags(int j, int k)

When an operator is executed, this function returns the flags for the k^{th} level of the j^{th} operand, using the same values as dcl_level_flags(k) (section 3.2).

int op_levels(int j)

When an operator is executed, this function returns the number of levels of the j^{th} operand, using the same values as dcl_levels (section 3.2).

char * op_macro(void)

When a macro function with arguments is about to be expanded, (*i.e.* when op_macro_call is triggered), this function returns the name of the macro function that is to be expanded. *Important:* this function only applies to macros with arguments.

int op_parened_operand(int j)

When an operator is executed, this function checks if the jth operand is an expression within a pair of parenthesis. If yes, function returns 1, otherwise 0.

void skip_macro_ops(int b)

This function gives a way to control whether op_ variables are effective on operators from macro expansion. If operators from a macro expansion need to be counted, call this function with zero as an argument, by default, CodeCheck acts like skip_macro_ops(1) has been called.

3.10 Preprocessor Variables

All predefined CodeCheck variables that have the prefix *pp_* refer to characteristics of preprocessor directives, *i.e.* lines that begin with the character #. Every preprocessor variable is initialized to zero at the start of execution, ***and again at the end of the scan of every preprocessor directive.***

pp_ansi	Set to 1 if a preprocessor feature is encountered that is new with the ANSI standard.
pp_arg_count	Set to the number of formal parameters found in a macro definition. (Use pp_empty_arglist to detect macro "functions" with zero parameters)
pp_arg_multiple	Set to 1 if a macro formal parameter is used more than once in the macro definition.
pp_arg_paren	Set to 1 if a macro formal parameter is used without being surrounded by parentheses.
pp_arg_string	Set to 1 if a macro formal parameter is found inside a string literal in the macro definition.
pp_arith	Set to 1 if a preprocessor directive requires arithmetic calculation.ifif
pp_assign	Set to 1 if a macro definition is a simple assignment.definedefine
pp_bad_white	Set to 1 if a non-space, non-tab whitespace character (*e.g.* vertical tab, form-feed, or backspace) is encountered within a preprocessor directive.
pp_benign	Set to 1 if a macro is redefined to be virtually identical to its previous definition.
pp_comment	Set to 1 if two tokens within a macro definition are separated only by a comment.
pp_const	Set to 1 if a macro is a manifest constant (it has no formal parameters and its body consists of a string, character, or numeric constant).

pp_defined	Set to 1 if the *defined* preprocessor function is encountered.
pp_depend	Set to 1 if undefundefundef is used on a macro that is used by other macros.
pp_elif	Set to 1 if the elifelifelif preprocessor directive is encountered.elifelifelif
pp_endif	Set to 1 if the endifendifendif preprocessor directive is encountered.endifendifendif
pp_empty_arglist	Set to 1 if the definition of a macro "function" has no formal parameters.
pp_empty_body	Set to 1 if the definition of a macro has no body.
pp_error	Set to 1 if the #error preprocessor directive is encountered.errorerror
pp_if_depth	Set to the new depth of conditional compilation whenever an #if, #ifdef, #ifndef, #else, #elif, or #endif directive is activated.
pp_include	After an #include directive has been read, but before the header is actually opened, this variable is set to one of the following values: 1: filename is in quotes, from a macro expansion, 2: filename is in quotes, ***not*** from a macro, 3: filename is in angle brackets, from a macro, 4: filename is in angle brackets, ***not*** from a macro. 5: filename is not enclosed (Metaware *only*), 6: filename is not enclosed (Vax VMS *only*).
pp_include_depth	Set to the new depth of file inclusion whenever an *includeincludeinclude* directive is executed, or an end-of-file in a header file is encountered. See also lin_source.
pp_include_white	Set to 1 if the filename in an *#include* directive has leading whitespace.
pp_keyword	Set to 1 if a macro name is a reserved ANSI or C++ keyword, (but *not* if it is an implementation-specific reserved keyword, *e.g.* near).

pp_length	Set to the length (in characters) of the body of a macro definition. Each occurrence of whitespace within the body of the macro counts as one character.
pp_lowercase	Set to 1 if the macro name in a macro definition is defined with any letters that are lowercase.
pp_macro	Set to the length in characters of a macro name when it is defined.
pp_macro_conflict	Set to 1 if a macro is defined differently in separate modules of a project.
pp_macro_dup	Set to 1 if a macro is defined in more than one file.
pp_not_ansi	Set to 1 whenever the preprocessor is used in a way that violates the ANSI standard.
pp_not_defined	Set to 1 if a preprocessor arithmetic expression (*e.g.* in an #if directive) uses an identifier that has not been defined as a macro.
pp_not_found	Set to 1 when an #include file could not be found.
pp_overload	Set to 1 if a variable name conflicts with a macro function name.
pp_paste	Set to 1 if the ANSI paste operator (##) is found in a macro definition.
pp_paste_failed	Set to 1 if the operands of the ANSI paste operator (##) could not be pasted together. According to ANSI, the result is undefined, and is therefore not portable.
pp_pragma	Set to 1 if a pragmapragmapragma preprocessor directive is encountered.
pp_recursive	Set to 1 if a recursive macro definition is found.
pp_relative	Set to 1 when an #include directive within a header file specifies a relative pathname.
pp_semicolon	Set to 1 if a macro definition ends with a semicolon.

pp_sizeof	Set to 1 if a preprocessor directive requires the use of a *sizeof* operator.
pp_stack	Set to 1 if a macro is redefined within a module (0 if the redefinition is benign).
pp_stringize	Set to 1 if the ANSI “stringize” operator (#) is found in a macro definition.
pp_sub_keyword	Set to 1 if the keyword in a preprocessor directive is itself a macro name.
pp_trailer	Set to 1 if a preprocessor line contains any nonwhite characters after the end of the directive and before the end of the line.
pp_undef	Set to 1 whenever undefundefundef is used.undef
pp_unknown	Set to 1 if a preprocessor directive is found with which CodeCheck is unfamiliar.
pp_unstack	Set to 1 if #undef is used to unstack multiply-defined macros.undefundefundef
pp_white_before	Set to the amount of whitespace (in characters) that precedes the # character in a preprocessor directive.
pp_white_after	Set to the amount of whitespace (in characters) that is found after the # character and before the keyword in a preprocessor directive.

Associated CodeCheck variables:

conflict_line	When pp_macro_conflict is triggered (when a macro definition conflicts with an earlier definition), this variable is set to the line number for the earlier definition. The file name is returned by the function conflict_file().

Associated CodeCheck functions:

char * conflict_file(void)

When pp_macro_conflict is triggered (when a macro definition conflicts with an earlier macro definition), this function returns the name of the file for the earlier definition. The line number is given by the variable conflict_line.

void define(char * name, char * body)

This function defines a macro with the specified name and body, just as though the definition had appeared in the source file. The macro may not have any arguments.

char * header_name(void)

When a header file is about to be opened with the #include directive, this function returns the filename. It may be used as a trigger in a CodeCheck rule. The event that trigggers this function occurs *before* pp_include is triggered.

char * header_path(void)

When a header file is about to be opened with the #include directive, this function returns the pathname to the directory in which the header was found. If the header path is the current directory, then header_path() returns zero. It may be used as a trigger in a CodeCheck rule. The event that trigggers this function occurs *before* pp_include is triggered.

int macro(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns the value 1 whenever a macro with the given name is about to be expanded.

int macro_defined(char *name)

This function inquires if the macro with specified name has been defined at the point where it triggers the rule containing the call to this function. If the macro has been found defined, the function returns 1, otherwise 0. Do not use this function in the rule triggered by prj_begin because at that time the macros have not been defined yet. Also this function is not for the use as a trigger.

char * op_macro(void)

When a macro function with arguments is about to be expanded, (*i.e.* when op_macro_call is triggered), this function returns the name of the macro function that is to be expanded. *Important:* this function only applies to macros with arguments.

void pp_error_severity(int s)

This function controls how to deal with the preprocessor directive #error. Normally, CodeCheck will quit execution after giving a fatal error message with the string following #error once an #error directive is encountered. This is also the way a majority of C/C++ compilers behave. However, there are some C/C++ compilers that allow users to use this directive just for displaying messages and proceed with the compilation. To make CodeCheck allow checking to continue on #error directives, call this function with INFO_PP as an argument. To make CodeCheck treat #error directives as fatal errors, call this function with ERROR_PP as an argument. INFO_PP and ERROR_PP are defined in file check.cch. They are defined as:

```
#define   INFO_PP   0

#define   ERROR_PP  1
```

CodeCheck acts like pp_error_severity(PP_ERROR) has been called by default.

char * pp_name(void)

When a macro is defined, this function returns the name of the macro.

int pragma(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns 1 whenever a pragma of the specified name is found.

void undefine(char * name)

This function undefines the macro with the specified name.

3.11 Project Variables

All CodeCheck predefined variables that have the prefix prj_ refer to characteristics of entire projects. These variables are initialized to zero at the start of execution, and all except prj_begin receive their values when the end of a project has been found. The special variable prj_begin is triggered just before a project is checked. If CodeCheck has been called for a single source file only, then this file is considered to be the entire project.

prj_aggr	Set to the number of external array, union, struct, or class variables in a project.
prj_array	Set to the number of external array elements in a project.
prj_begin	Set to 1 at the start of a project.
prj_com_lines	Set to the number of pure comment lines in a project.
prj_conflicts	Set to the number of conflicting macro definitions found in a project.
prj_decisions	Set to the number of binary decision points in a project.
prj_end	Set to 1 when a project has been completely read.
prj_exec_lines	Set to the number of executable lines in a project.
prj_functions	Set to the number of functions defined in a project. C++ class member functions and functions that are declared but never defined are not counted.
prj_globals	Set to the number of external variables in a project.
prj_H_operands	Set to the number of Halstead operands found in a project before macro expansion.
prj_H_operators	Set to the number of Halstead operators found in a project before macro expansion.

prj_headers	Set to the number of header (.h) files read. Each header file is only counted once.
prj_high	Set to the number of high-level statements found in a project.
prj_low	Set to the number of low-level statements found in a project.
prj_macros	Set to the number of macros defined in a project.
prj_members	Set to the number of members of union, struct, or class global tags (*i.e.* declared with file scope).
prj_modules	Set to the number of source (.c) files read.
prj_nonexec	Set to the number of non-executable statements found in a project.
prj_operands	Set to the number of operands found in a project, before macro expansion.
prj_operators	Set to the number of operators found in a project before macro expansion.
prj_simple	Set to the number of simple external variables (char, short, long, int, unsigned, float, or double) in a project.
prj_tokens	Set to the number of tokens found in a project before macro expansion.
prj_total_lines	Set to the total number of lines in a project.
prj_u_operands	Set to the number of unique operands found in a project, before macro expansion.
prj_u_operators	Set to the number of unique operators found in a project before macro expansion.
prj_uH_operands	Set to the number of unique Halstead operands found in a project, before macro expansion.
prj_uH_operators	Set to the number of unique Halstead operators found in a project before macro expansion.
prj_unused	Set to the number of unused external variables in a project.

prj_warnings	Set to the number of CodeCheck warnings issued for this project.
prj_white_lines	Set to the number of whitespace lines in a project.

Associated CodeCheck functions:

char * prj_name(void)

This function returns the name of the project that is currently being checked, *i.e.* the name of the .ccp file that identifies the component modules of a project to CodeCheck. If there is no project file, then this function returns the name of the first source file in the command-line.

3.12 Statement Variables

All predefined CodeCheck variables that have the prefix stm_ refer to characteristics of C statements, broadly interpreted. CodeCheck defines a C statement as any of these entities: a declaration, a type definition, a simple executable statement that ends with a semicolon, a compound statement that begins and ends with braces, or a goto, for, while, switch, if or do statement. Every statement variable is initialized to zero at the start of execution, ***and again at the end of the scan of every statement.*** CodeCheck evaluates statements recursively, so that variables that refer to statements that contain statements are correctly set.

Note that simple C statements are conceptually very different from lines of C code, even though most programmers place at most one statement on each line. It is nevertheless possible to have more than one statement per line, or more than one line per statement. For CodeCheck variables that apply to *lines of code*, see the Section 3.7.

Several statement variables are set to a value which indicates the kind of statement that has been found. These values are defined as manifest constants in the CodeCheck header file check.cch. The constants are:

```
#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
```

* ? A compound statement is a group of statements surrounded by curly braces.

stm_aggr	Set to the number of array, union, struct, or class variables declared in a compound statement.
stm_array	Set to the number of local array elements declared in a compound statement.
stm_bad_label	Set to 1 whenever a label or list of labels is found that is not attached to any statement.
stm_cases	Set to the number of case labels attached to this statement (includes the default label).
stm_catchs	Set to the number of exception handler's in a try-block.
stm_container	Set to the kind of high-level statement that *contains* the current statement (IF through COMPOUND).
stm_cp_begin	When the open curly brace of a compound statement has been found, this variable is set to the context of the compound statement (IF through COMPOUND).
stm_cp_assign	Set to the number of compound assignment operators (*e.g.* +=, |=) found in a low-level statement.
stm_depth	Set to the logical depth of a statement, *i.e.* its nesting level within if, for, while, and do statements.
stm_end	Set to 1 when the end of a statement has been found.
stm_end_tryblock	Set to 1 when the end of a try-block is found, at the closing brace of last catch clause.
stm_goto	Set to 1 if a goto enters a block that has automatic or register variable initializers.
stm_is_comp	When the close curly brace of a compound statement has been found, this variable is set to the context of the compound statement (IF through COMPOUND).
stm_if_else	Set to 1 if there is a matching else statement for an if statement, set at the end of if statement.
stm_is_expr	Set to 1 if this is an expression statement.

stm_is_high	Set to 1 if this is a compound, selection, or iteration statement (IF through COMPOUND).
stm_is_iter	Set to 1 if this is an iteration statement (WHILE, DO, or FOR).
stm_is_jump	Set to 1 if this is a jump statement (BREAK through GOTO).
stm_is_low	Set to 1 if this is an expression or jump statement (EXPRESSION, BREAK, CONTINUE, RETURN or GOTO). This variable will also trigger on a local C++ declaration that has an initializer.
stm_is_nonexec	Set to 1 if this is a local declaration. This does ***not*** trigger on a local C++ declaration that has an initializer.
stm_is_select	Set to 1 if this is a selection statement (IF, ELSE, or SWITCH).
stm_kind	Set to the kind of the current statement (IF through EMPTY).
stm_labels	Set to the number of ordinary labels (not including case or default labels) attached to this statement.
stm_lines	Set to the number of lines in a statement.
stm_locals	Set to the number of local variables declared in a compound statement.
stm_loop_back	Set to 1 when a *goto* statement is found that transmits control back to a previous label.
stm_members	Set to the number of local union, struct, or class members declared in a compound statement.
stm_need_comp	Set to 1 if a statement contained by if, else, while, do and for is not a compound statement.
stm_never_caught	Set to 1 if an exception handler (catch) will never be reached because the type is shadowed by the previous handler(s).

stm_no_break	Set to 1 if the current statement is a case statement in a switch, and the previous case did not terminate with a transfer of control (*e.g*. a break or return).
stm_no_default	Set to 1 if the current statement is a switch without a default case.
stm_operands	Set to the total number of operands found in a statement, before macro expansion.
stm_operators	Set to the total number of standard C operators found in a statement.
stm_relation	Set to the number of Boolean relational operators found in a statement.
stm_return_paren	Set to 1 if a return has a value that is ***not*** enclosed in parentheses.
stm_return_void	Set to 1 if: (1) a return has no value in a function declared to return a non-void type, (2) if a function has no return statement but requires a returned value, or (3) if a return has a value in a function declared to return void.
stm_semicolon	Set to 1 if a suspicious semicolon is found, *e.g*. in the statement while(x); .
stm_simple	Set to the number of local simple variables (char, short, long, int, unsigned, float, or double) declared in a compound statement.
stm_switch_cases	Set to the number of cases found in a switch.
stm_tokens	Set to the number of tokens found in a statement.
stm_unused	Set to the number of local variables declared in a compound statement but never used.

3.13 Structure and Class Variables

All predefined CodeCheck variables that have the prefix tag_ refer to characteristics of the definitions of classes, structs, unions, and enums. Every CodeCheck tag variable is initialized to zero at the start of execution, ***and again at the end of the scan of every tag definition.*** CodeCheck evaluates tag definitions recursively, so that variables that refer to tags that contain tag definitions are correctly set. Except when it is clear from context, the C++ term "class" in the following may be taken to mean class, struct, or union.

tag_abstract	Set to 1 if this is an abstract class (*i.e.* it has at least one pure virtual member function).
tag_anonymous	Set to 1 if this tag is anonymous (has no tag name).
tag_base_access	Set to 1 if a base class does not have an explicit access specifier (public, protected, or private).
tag_bases	Set to the number of direct base classes declared for this class.
tag_begin	Set to 1 when the left curly brace of a tag definition is found.
tag_classes	Set to the number of tags defined within this class. This count includes enum tags and excludes anonymous (unnamed) tags.
tag_constants	Set to the number of enumerated constants defined in this class.
tag_constructors	Set to the number of constructors declared in this class.
tag_distance	Set to 1 for a near class, 2 for a far class, 3 for a huge class, and 4 for an export class.
tag_end	Set to 1 at the end of a tag definition.
tag_fcn_friends	Set to the number of friend *functions* declared in this class.

tag_friends	Set to the number of friend *classes* declared in this class.
tag_functions	Set to the number of C++ member functions declared in this class.
tag_global	Set to 1 if this tag has file scope.
tag_has_assign	Set to 1 if this class has an operator= function.
tag_has_copy	Set to 1 if this class has an explicit copy constructor.
tag_has_default	Set to 1 if this class has an explicit default constructor (a constructor with no parameters).
tag_has_destr	Set to 1 if this class has an explicit destructor.
tag_hidden	Set to 1 when this local tag definition hides another at file or local scope.
tag_kind	Set to 1 for an enum, 2 for a union, 3 for a struct, or 4 for a class.
tag_lines	Set to the number of lines in the tag definition.
tag_local	Set to 1 if this tag is local (*i.e.* defined within a function definition).
tag_mem_access	Set to 1 if the first member of a class, struct, or union does not have an explicit access label (public, protected, or private).
tag_members	Set to the number of data members (also known as instance variables) declared in this tag. *Not counted* are: C++ member functions, enumerated constants (except in enum tags), typedef names, components of nested tags (if named), and components of base classes. Use the function tag_components() for detailed counts.
tag_nested	Set to 1 if this tag is nested (*i.e.* defined within a class definition).
tag_operators	Set to the number of operator functions declared in this tag.

tag_private	Set to the total number of identifiers declared in this tag that have private access.
tag_protected	Set to the total number of identifiers declared in this tag that have protected access.
tag_public	Set to the total number of identifiers declared in this tag that have public access.
tag_static_fcn	Set to the number of static member functions declared in this class.
tag_static_mem	Set to the number of static data members declared in this class.
tag_template	Set to the number of template parameters if this is a template definition.
tag_tokens	Set to the number of tokens in the tag definition.
tag_types	Set to the number of typedef names and tag names defined within this tag.

Associated CodeCheck variables:

lex_invisible	Set to 1 when an *unscoped* tag name refers to a tag whose definition is nested within another tag definition. Such a tag name is visible in C and all versions of C++ prior to 3.0, but is invisible in C++ 3.0.

Associated CodeCheck functions:

char * class_name(void)

When lin_within_class is non-zero (in a C++ class *or* class member function definition), this function returns the class name. *Note:* this will be different from tag_name() when an enum is currently being defined within a class definition, or when any tag is locally defined within a class member function.

int mod_class_lines(int index)

This function returns the total number of lines in each named class, struct, or union defined in the module, indexed by its order within the module. These lines *include* lines in definitions of member functions that are outside the class definition. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

char * mod_class_name(int index)

This function returns the name of each named class, struct, or union defined in the module, indexed by its order within the module. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

int mod_class_tokens(int index)

This function returns the number of tokens in each named class, struct, or union defined in the module, indexed by its order within the module. Tokens in definitions of member functions defined outside the class definition are *included* in the token count. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

char * tag_name(void)

Returns the name of the tag that is currently being defined. *Note:* this will be different from class_name() when an enum is currently being defined within a class definition, or when any tag is locally defined within a class member function.

int tag_baseclass_access(int j)

This function returns the access specifier type of jth base class of the class being checked. Use tag_bases to obtain the number of the number of base class. If the value of augument exceeds the actual number of base class, the result is undefined. Otherwise, it returns 0 for public base classes, 1 for protected base classes and 2 for private base classes.

int tag_baseclass_kind(int j)

This function returns the kind of jth base class of the class being checked. Use tag_bases to obtain the number of the number of base class. If the value of

augument exceeds the actual number of base class, the result is undefined. Otherwise, the returned value has same meaning as the value of tag_kind.

char *tag_baseclass_name(int j)

This function returns the name of jth base class of the class being checked. Use tag_bases to obtain the number of the number of base class. If the value of augument exceeds the actual number of base class. 0 is returned as result.

int tag_components(int kind, int access)

Returns the number of tag components of the specified kind and access. The possible values of kind and access are:

kind	**access**
0: constants	0: any
1: members	1: public only
2: functions	2: protected only
3: types	3: private only
4: base classes	

Chapter 4: CodeCheck Functions

CodeCheck has many useful intrinsic functions, listed alphabetically below. The type char* appears in a parameter list when the argument can be either a string literal or a CodeCheck function that returns a string. *Unless otherwise stated, these functions cannot be used as rule triggers.*

CodeCheck has a variety of useful intrinsic functions, as described here with C prototypes. These CodeCheck functions are grouped here by category. For an alphabetical listing, consul the index under the heading “function”.

Details of the special storage class statistic, which is used by all of CodeCheck’s statistical functions, are covered in section 2.6.4.

The type char* appears in a parameter list when the argument can be either a string literal or a CodeCheck function that returns a string.

4.1 General Functions

`void advise ( int on_off )`

Enable and/or Disable CodeCheck Internal Warning Messages from Console Output *Stream*.

int exec(char * program, char * arg1, char * arg2, ...)

This function executes an operating system shell command. First the shell command is constructed by concatenating all the arguments together, separated by spaces. The resulting command string is then executed by the shell by means of a call to the ANSI standard system() function.

void exit(int n)

This function causes an immediate exit from CodeCheck, returning the error number *n* to the operating system.

void fatal(int n, char * message)

This function prints an error number and message to stderr and then exits CodeCheck, returning the error number *n* to the operating system.

char * fcn_name(void)

This function returns the name of the function that is currently being checked.

char * file_name(void)

This function returns the name of the file that is currently being checked. See also: mod_name(), prj_name().

void force_include(char *header_name, int header_type, int add_or_del)

Function **force_include** acts similar to the command option /FI of MSVC++ which force's a specified header file to be included at the beginning of a module when an explicit #include directive is used. **Header_name** specifies the name of the file to be included. **Header_type** specifies the file should be included as user header file or system header file. **Add_or_del** decides if the file should be added to or removed from the list of files to be included in this way, remove the file from list if this parameter has value 0, otherwise add file to the list.

This function decides if the file name should stay in the candidate list. At the beginning of a module, CodeCheck will go through the file list and include them. Files can be added to or removed from the file list any time. However, the effect only shows at the beginning of next module.

int included(char * name)

Returns 1 if the argument is the name of a header file that has been fully included in the current module.

char * header_name(void)

When a header file is about to be opened with the #include directive, this function returns the filename. It may be used as a trigger in a CodeCheck rule.

char * header_path(void)

When a header file is about to be opened with the #include directive, this function returns the pathname to the directory in which the header was found. If the header path is the current directory, then header_path() returns zero. It may be used as a trigger in a CodeCheck rule.

char * line(void)

This function returns the current input line as a null-delimited string. This string does not end with a newline character.

char * mod_name(void)

This function returns the name of the module that is currently being checked. A “module” is a C source file and all of its header files. Its name is the name of the first source file in the module. See also: file_name(), prj_name().

int option(char c)

This function returns 1 if the command line option specified by *c* is in effect (*i.e.* has been specified by the user), otherwise it returns 0.

void remove_path(); (void)

This function will make a path that was set for searching of included header files invalid. Only the least recent set path is removed from the list. The path can only be the one set by function call set_str_option('I', ...). If there is no path left in the including path list, this function has no effect.

char * prj_name(void)

This function returns the name of the project that is currently being checked, *i.e.* the name of the .ccp file that identifies the component modules of a project to CodeCheck. If there is no project file, then this function returns the

name of the first source file in the command-line. See also: mod_name(), file_name().

void set_header_optS(char *header_name, int header_type, int check_option, int pass_along)

Normally, whether rules are to be applied to system header files is controlled by command option –S. In the middle of the process, it is possible to change option –S by calling function set_option('S', option) within rules. However, it is very difficult to change the option for a specific header file to be included.

This function allows option -S to be set to a different value just for specified header file. The function takes 4 parameters.

header_name, A string that specifies name of the header file this function is to be called upon, it must have the same format as the actual header file name used in #include directive, character by character.

header_type Set to 1 if the header file is a user header file, *i.e.* included within a pair of double quotes. 2 if the header file is a system header file, *i.e.* included within pair of angle brackets.

check_option This takes the same value as command option –S, it must be 0,1,2 and 3.

pass_along If this option has value 0, the effect of option –S set for this file will not be passed along into the header files included directly or indirectly within this header file. In the case of a non-zero argument the option –S set for this header file will be effective to the header files included by the calling header file directly or indirectly unless the nested header files set their own option –S by calling this function.

void set_option(char c, int n)

This function assigns the value *n* to the command-line option specified by *c*. For example, when set_option('B',1) is executed by CodeCheck, all succeeding rules are evaluated as if the user had specified the **-B** option on the command-line that invoked CodeCheck. The **–K** option cannot be changed with this function — it must be set on the command-line.

void set_str_option(char c, char * name)

This function assigns the string name to the command-line option specified by c. For example, when set_str_option('I',"/new/hdrs") is executed by CodeCheck, all succeeding rules are evaluated as if the user had specified the **–I/new/hdrs** option on the command-line that invoked CodeCheck.

char * str_option(char c)

This function returns the string value of the command-line option specified by *c*. For example, when str_option('L') will return the name of the listing file, as it was specified in the command-line with option **–L**. If the option is not in the command-line, then str_option will return an empty string.

int test_needed(char * name, ...)

This function is to be used *only* as a trigger in a CodeCheck rule. It is designed to detect the circumstance in which a function (for example malloc) is called but its return value is not tested (for example, compared to NULL). The function test_needed returns 1 if a function with a specified name is called and either: (a) the current expression is not within an *if*- or *while*-condition, or (b) the next statement is not an *if*- or *switch*-statement. The argument list for may be a list of function names. The name of the triggering function may be obtained with prev_token().

char * time_stamp(void)

This function may be used for time-stamping CodeCheck reports. It returns a time-and-date string.

4.2 Lexical Functions

int find_root(char * symbol)

Find root base type of name symbol within current local scope. Useful for advanced symbol table lookup algorithms. Can be used to determine if a scope-name is valid. See check.cch for values returned, zero is returned if symbol unknown.

int find_scoped_root(char *scope-name, char * symbol-name).

Find root of symbol using an explicit scope name. Very useful for testing the presence of members within named classes. Returns zero if symbol not found within explicit scope.

int identifier(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns the value 1 whenever an identifier (a variable or function name) has been encountered that matches the given string. See also: keyword(), macro(), token(), prev_token().

void ignore(char * name, ...)

This function causes the CodeCheck lexical analyzer to ignore any identifier or keyword that matches one of the argument names. Every argument must be a string.

int keyword(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns the value 1 whenever a keyword has been encountered that matches the given string. See also identifier(), token(), prev_token(), macro().

char next_char(void)

This function returns the lexical analyzer's lookahead character: the character in the CodeCheck input stream that immediately follows the current token. This function may not be used as a rule trigger.

char * next_token(void)

Returns string pointer to look-ahead token in source stream from current position. Return value may be NULL near end of line. Used with prev_token(), token(), and next_char().

int prefix(char * str)

This function returns 1 if the identifier currently being defined begins the letters in str, otherwise 0. Each subsequent call to prefix **within the same rule** will start looking for the specified prefix immediately *after* the last successfully recognized prefix. Thus prefix can be used to parse sequences of prefixes, from left to right. See also root() and suffix().

char * prev_token(void)

This function returns the previous token that has being parsed by CodeCheck. The token is in string form. See also keyword(), identifier(), macro(), and token().

void skip_nonansi_indent(char c);

This function will cause CodeCheck to skip indentifiers that start with non-ansi characters, e.g. '@', '$', or '`'. The value of the parameter can only be one of the previously mentioned characters, all other values will have no effect.

char * root(void)

Returns the root of an identifier currently being defined after application of either of the functions prefix and/or suffix. For example, after calling prefix("foo_") on the identifier foo_bar, the function root() will return the string "bar".

char * stm_unused_name(int k)

When there are one or more unused variables in a block, then this function returns the name of the each unused variable, for $0 = k < stm\_unused$.

int suffix(char * str)

This function returns 1 if the identifier currently being defined ends with the letters in str, otherwise 0. Each subsequent call to suffix **within the same rule** will start looking for the specified suffix *before* the last successfully recognized suffix. Thus suffix can be used to parse sequences of suffixes, from right to left. See also prefix() and root().

char * token(void)

This function returns the current token that has being parsed by CodeCheck. The token is in string form. See also keyword(), identifier(), and macro().

4.3 Preprocessor Functions

char * conflict_file(void)

When pp_macro_conflict is triggered (when a macro definition conflicts with an earlier definition), this function returns the name of the file for the earlier definition. The line number is given by the variable conflict_line.

void define(char * name, char * body)

This function defines a macro with the specified name and body, just as though the definition had appeared in the source file. The macro may not have any arguments.

int macro(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns the value 1 whenever a macro with the given name is about to be expanded. See also: keyword(), identifier(), token(), prev_token().

int macro_defined(char *name)

This function inquires if the macro with the specified name has been defined at the point where it triggers the rule containing the call to this function. If the macro has been defined, the function returns 1, otherwise 0. Do not use this function in the rule triggered by prj_begin because at that time the macros have not been defined yet. Lastly, this function is not for the use as a trigger.

int no_undef(char * identifier)

This function returns 1 if the specified identifier has ***not*** been previously #undefined in the source file. It may not be used as a trigger in a CodeCheck rule.

char * op_macro(void)

When a macro function with arguments is about to be expanded, (*i.e.* when op_macro_call is triggered), this function returns the name of the macro func-

tion that is to be expanded. *Important:* this function only applies to macros with arguments.

void pp_error_severity(int s)

This function controls how to deal with the preprocessor directive #error. Normally, CodeCheck will quit execution after giving a fatal error message with the string following #error once an #error directive is encountered. This is the way a majority of C/C++ compilers behave. However, there are some IBM C/C++ compilers that allow users to use this directive just for displaying certain 'error' messages and continue normally. To make CodeCheck allow checking to continue on #error directives, call this function with INFO_PP as argument. To make CodeCheck treat #error directives as fatal errors, call this function with ERROR_PP as an argument. INFO_PP and ERROR_PP are defined in file check.cch. By default, CodeCheck acts like pp_error_severity(PP_ERROR) has been called.

void pp_if_search(int)

Enable GNU-GCC #if (types) pre-processor method. Open-System embedded compiler support. GNU-GCC allows #if test on actual types in addition to simple macro testing. Default for this feature is off [zero-value], on [nonzero-value]

char * pp_name(void)

When a macro is defined, this function returns the name of the macro.

int pragma(char * name)

This function is designed to be used as a trigger in a CodeCheck rule. It returns 1 whenever a pragma of the specified name is found.

void undefine(char * name)

This function undefines the macro with the specified name.

4.4 Declarator Functions

char * conflict_file(void)

When dcl_conflict is triggered (when a declaration conflicts with an earlier declaration), this function returns the name of the file for the earlier declaration. The line number is given by the variable conflict_line.

char * dcl_base_name(void)

This function returns the name of the base type of the current declarator. If the base type is a typedef name then the typedef name is returned. If the base type is an enum, union, struct, or class, then the tag name is returned.

int dcl_level(int level)

Set to an integer which identifies the kind of the specified level (*function returning…*, *reference to…*, *pointer to…*, or *array of…*) for the current declarator. The number of levels for the current declarator is given by dcl_levels, which is zero for simple variables. The kinds are defined as manifest constants in the CodeCheck header file check.cch. These constants are:

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

int dcl_level_flags(int level)

Set to an integer which identifies all of the type qualifiers (*e.g.* const) of the specified level (*pointer to…*, *array of…*, *function returning…*, or *reference to…*) of the current declarator. The number of levels for the current declarator is given by dcl_levels, which is zero for simple variables. The last level always refers to the base type of the declarator. The level flags are defined as manifest constants in the CodeCheck header file check.cch. These constants and an example are given in Section 3.2.

char * dcl_name(void)

If CodeCheck is scanning a declarator, then this function returns the name of the current declarator, otherwise 0.

char *dcl_scope_name(void)

This function returns the class scope name right before the declarator. If the declarator is not scoped, the function returns 0.

void new_type(char * name, int type)

This function informs CodeCheck of the existence of a nonstandard keyword for a base type. The first argument for new_type() should be the new keyword itself, in quotes. The second argument should be any of the possible values of dcl_base (which are defined as manifest constants in the standard CodeCheck header check.cch) *except* DEFINED_TYPE. If the value is one of these:

```
#define EXTRA_INT_TYPE       6  //  e.g. Macintosh comp type
#define EXTRA_UINT_TYPE     11
#define EXTRA_FLOAT_TYPE    15  //  e.g. Macintosh extended type
#define EXTRA_PTR_TYPE      21  //  e.g. Microsoft _segment type
```

then CodeCheck will treat the new keyword as a new unique base type. If it is any other value then the keyword will be considered a synonym for the specified C type. Consult check.cch for the complete list of base types.

4.5 C++ Class Functions

char * class_name(void)

When lin_within_class is non-zero (in a C++ class *or* class member function definition), this function returns the class name.

int mod_class_lines(int index)

This function returns the total number of lines in each named class, struct, or union defined in the module, indexed by its order within the module. These lines *include* lines in definitions of member functions that are outside the class definition. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

char * mod_class_name(int index)

This function returns the name of each named class, struct, or union defined in the module, indexed by its order within the module. The index is

zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

int mod_class_tokens(int index)

This function returns the number of tokens in each named class, struct, or union defined in the module, indexed by its order within the module. Tokens in definitions of member functions defined outside the class definition are *included* in the token count. The index is zero-based: the first tag has index 0 and the number of classes is given by mod_classes.

int tag_baseclass_access(int j)

This function returns the access specifier type of jth base class of the class being checked. Use tag_bases to obtain the number of the number of base class. If the value of augument exceeds the actual number of base class, the result is undefined. Otherwise, it returns 0 for public base classes, 1 for protected base classes and 2 for private base classes.

int tag_baseclass_kind(int j)

This function returns the kind of jth base class of the class being checked. Use tag_bases to obtain the number of the number of base class. If the value of augument exceeds the actual number of base class, the result is undefined. Otherwise, the returned value has same meaning as the value of tag_kind.

char *tag_baseclass_name(int j)

This function returns the name of jth base class of the class being checked. Use tag_bases to obtain the number of the number of base class. If the value of augument exceeds the actual number of base class. 0 is returned as result.

char * tag_name(void)

Returns the name of the tag that is currently being defined. *Note:* this will be different from class_name() when an enum is currently being defined within a class definition, or when any tag is locally defined within a class member function.

int tag_components(int kind, int access)

Returns the number of tag components of the specified kind and access. The possible values of kind and access are:

kind	**access**
0: constants	0: any
1: members	1: public only
2: functions	2: protected only
3: types	3: private only
4: base classes	

4.6 Operator Functions

int op_base(int j)

When an operator is executed, this function returns the base type of the j^{th} operand, using the same values as dcl_base (section 3.2).

char * op_base_name(int j)

If the base type of the j^{th} operand of an executable operator is a tag (enum, union, struct, or class) or typedef name, then this function returns the tag or typedef name as a character string.

int op_bitfield(int j)

Returns 1 if the base type of the j^{th} operand of an executable operator is a bitfield, otherwise zero.

char * op_function(void)

When op_open_funargs, op_close_funargs, or op_call is triggered, this function returns the name of the function that is to be declared or called. When an expression has to be evaluated in order to find which function to call, *e.g.* through a function pointer, then this function returns an empty string.

int op_level(int j, int k)

When an operator is executed, this function returns the kind of the k^{th} level of the j^{th} operand, using the same values as dcl_level(k) (section 3.2).

int op_level_flags(int j, int k)

When an operator is executed, this function returns the flags for the k^{th} level of the j^{th} operand, using the same values as dcl_level_flags(k) (section 3.2).

int op_levels(int j)

When an operator is executed, this function returns the number of levels of the j^{th} operand, using the same values as dcl_levels (section 3.2).

char * op_macro(void)

This function returns the name of the macro function as it is about to be expanded. It is triggered at the same time as op_macro_call. Note that this function only applies to macros with arguments. See also op_function().

int op_parened_operand(int j)

When an operator is executed, this function checks if the jth operand is an expression within a pair of parenthesis. If yes, function returns 1, otherwise 0.

void skip_macro_ops(int b)

This function gives a way to control whether op_ variables are effective on operators from macro expansion. If operators from a macro expansion need to be counted, call this function with a non-zero integer as argument. Otherwise, pass 0 as argument. CodeCheck acts like skip_macro_ops (1) has been called by default.

By default, op_ variables will not be effective on the operators derived from macro expansion. This function provides a way let user control if op_ variables should be effective on operators derived from macro expansion. When value 0 is passed into function as actual argument, CodeCheck will not ignore the operators of this kind while setting up op_ variables. When a non-zero value is passed as actual argument, op_ variables will not be set for the operators derived from macro expansion.

4.7 Character Functions

int isalpha(int ch)

Returns 1 if ch is an alphabetic character. Precisely the same as the ANSI function of the same name.

int isdigit(int ch)

Returns 1 if ch is a decimal digit. Precisely the same as the ANSI function of the same name.

int islower(int ch)

Returns 1 if ch is a lower case letter. Precisely the same as the ANSI function of the same name.

int isupper(int ch)

Returns 1 if ch is an upper case letter. Precisely the same as the ANSI function of the same name.

int tolower(int ch)

If ch is an upper case letter, then this function returns the lower case version of ch, otherwise it returns ch. Precisely the same as the ANSI function of the same name.

int toupper(int ch)

If ch is an upper case letter, then this function returns the lower case version of ch, otherwise it returns ch. Precisely the same as the ANSI function of the same name.

4.8 String Functions

int all_digit(char * s)

This function returns 1 if string pointed to by s consists only digits ('0'-'9'). Otherwise it returns 0.

int all_lower(char * s)

This function returns 1 if string pointed to by s consists only lower case letters ('a'-'z'). Otherwise it returns 0.

int all_upper(char * s)

This function returns 1 if string pointed to by s consists only upper case letters ('A'-'Z'). Otherwise it returns 0.

float atof(char *)

This function converts the initial portion of a string to a float. It is identical to ANSI function atof() except that a fatal error is generated if its argument is null pointer.

int aoti(char *)

This function converts the initial portion of a string to an integer. It is identical to ANSI function atoi() except that a fatal error is generated if its argument is null pointer.

char * strcat(char * s1, char * s2)

This function is identical to ANSI function strcpy(). It appends a copy of the string pointed to by s2 (including the terminating null character) to the end of the string pointed by s1. The initial character of s2 overwrites the null character at the end of s1. It returns the value of s1. If either of its arguments are null pointers, a fatal error is generated.

char * strchr(char * s, int c)

This function is identical to ANSI function strchr(), it locates the first occurrence of c (converted to a char) in the string pointed to by s. The

terminating null character is considered to be part of the string. It returns a pointer to the located character or 0 if the character does not exist in the string. The first argument is a null pointer, a fatal error is generated.

int strcmp(char * s1, char * s2)

This function is identical to ANSI function strcmp(), it compares the string pointed to by s1 to string pointed to by s2. It returns an integer greater than, equal to, or less than 0, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2. If either of arguments are null pointers, a fatal error is generated.

char * strcpy(char * s1, char * s2)

This is identical to ANSI function strcpy(), it copies the string pointed to by s2(including the terminating null character) into the array pointed to by s1. It returns the value of s1. If either of the arguments are null pointers, a fatal error is generated.

int strcspn(char * s1, char * s2)

This function is identical to ANSI function strcspn(), it computes the length of the maximum initial segment of the string pointed by s1 which consists entirely of characters not from string pointed to by s2. It returns the length of the segment. If either of arguments are null pointers, a fatal error is generated.

int strequiv(char * s1, char *s2)

This function return 1 if the two argument string pointed to by s1 and s2 are the same (not case sensitive). If they differ, or if either pointer is null, then strequiv returns 0.

int strlen(char *s)

This function is identical to ANSI function strlen(). It computes the length of the string pointed to by s. It returns a number of characters that precedes the terminating null character. If its argument is a null pointer, a fatal error is generated.

char * strncat(char *s1, char * s2, int n)

This function is identical to ANSI function strncat(). It appends no more than n characters (a null character and characters that follow it are not appended) from array pointed to by s2 to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result. It returns the value of s1. If either of its first two arguments are null pointers, a fatal error is generated.

int strncmp(char * s1, char * s2, int n)

This function is identical to ANSI function strncmp(), it compares not more than n characters (characters that follow a null character are not compared) from the array pointed to by s1 to array pointed to by s2. It returns an integer greater than, equal to , or less than 0, accordingly as the possibly null-terminated array pointed to by s1 is greater than, equal to, or less than the possibly null-terminated array pointed to by s2. If either of its first two arguments are null pointers, a fatal error is generated.

char * strncpy(char * s1, char * s2, int n)

This function is identical to ANSI function strncpy(), it copies not more than n characters (characters that follow a null character are not copied) from the array pointed to by s2 to array pointed to by s1. It returns the value of s1. If either of its first two arguments are null pointers, a fatal error is generated.

char * strpbrk(char *s1, char * s2)

This function is identical to ANSI function strpbrk(). It locates the first occurrence in the string pointed to by s1 of any character from the string pointed to by s2. It returns a pointer to the character, or 0 if no character from s2 exists in s1. If either of its arguments are null pointers, a fatal error is generated.

char * strrchr(char * s, int c)

This function is identical to ANSI function strrchr(). It locates the last occurrence of c (converted to a char) in the string pointed by s. The terminating null character is considered to be part of the string. It returns a pointer to the character, or 0 if c does not exist in the string. If the argument string is null pointer, a fatal error is generated.

int strspn(char * s1, char * s2)

This function is identical to ANSI function strspn(). It computes the length of the maximum initial segment of the string pointed to by s1, which consists entirely of character from the string pointed to by s2. It returns the length of the segment. If either of its arguments are null pointers, a fatal error is generated.

char * strstr(char * s1, char * s2)

This function is identical to ANSI function strstr(). It locates the first occurrence in the string pointed to by s1 of the sequence of characters (excluding the terminating null character) in the string pointed to by s2. It returns a pointer to the located string or 0 if the string is not found. If s2 points to a string with zero length, it returns s1. If either of its arguments are null pointers, a fatal error is generated.

4.9 Mathematical Functions

float log2(float)

This function returns the logarithm (base 2) of its argument.

float pow(float x, float y)

This function returns *x* raised to the power *y*.

float sqrt(float)

This function returns the square root of its argument.

4.10 Statistical Functions

float corr(statistic x, statistic y)

This function returns the correlation of its two argument variables. The argument variables must have the same number of cases. Pearson's product-moment correlation is returned.

void histogram(statistic x, int min, int max, int bins)

This function prints a histogram of its argument variable on stdout, using bins equal-width cells for all values between min and max. Every cell of the histogram counts the number of values observed that are greater than or equal to its lower bound, and less than its upper bound. If the last three parameters (min, max, bins) are zero, CodeCheck will use appropriate values based on the characteristics of the given statistic. Each cell of the histogram is labeled with its lower bound.

float maximum(statistic x)

This function returns the maximum (largest observed value) of its statistical argument variable.

float mean(statistic x)

This function returns the mean (arithmetic average) of its statistical argument variable.

float median(statistic x)

This function returns the median (*i.e.* the 50^{th} percentile) of its statistical argument variable.

float minimum(statistic x)

This function returns the minimum (smallest observed value) of its statistical argument variable.

float mode(statistic x)

This function returns the mode (most frequently observed value) of its statistical argument variable.

int ncases(statistic x)

This function returns the number of cases recorded for its statistical argument variable.

float quantile(statistic x, int k, int n)

This function returns the $\mathbf{k}^{th}$ **n**-tile of its statistical argument variable. To give two examples, quantile(x,95,100) returns the 95th percentile of x, while quantile(y,3,4) returns the third quartile of y.

void reset(statistic x)

This function resets its statistical argument variable. All recorded cases are erased, and the case count is reset to zero.

float stdev(statistic x)

This function returns the standard deviation of its statistical argument variable.

float variance(statistic x)

This function returns the variance of its statistical argument variable.

4.11 Input/Output Functions

int eprintf(char * format, ...)

This function behaves exactly as function printf() except that the output is on stderr instead of stdout.

int fclose(FILE * stream)

This function is identical to the ANSI standard fclose function. Do **not** include the header file stdio.h in the rule file — CodeCheck uses its own internal declarations for the standard IO functions.

FILE * fopen(char * filename)

This function is identical to the ANSI standard fopen function. Do **not** include the header file stdio.h in the rule file — CodeCheck uses its own internal declarations for the standard IO functions.

int fprintf(FILE * stream, char * format, ...)

This function is similar to the ANSI standard fprintf function. Do **not** include the header file stdio.h in the rule file — CodeCheck uses its own internal declarations for the standard IO functions.

int fscanf(FILE * stream, char * format, ...)

This function is similar to the ANSI standard fscanf function. The argument variables may be of type int, float, char, char[], or char*. All the usual formatting conventions are supported. Do **not** include the header file stdio.h in the rule file — CodeCheck uses its own internal declarations for the standard IO functions.

int printf(char * format, ...)

This function is similar to the ANSI standard printf function. The argument variables may be of type int, float, char, or char*. All the usual formatting conventions are supported *except* the asterisk notation and the %n format. Do **not** include the header file stdio.h in the rule file — CodeCheck uses its own internal declarations for the standard IO functions.

int scanf(char * format, ...)

This function is similar to the ANSI standard scanf function. The argument variables may be of type int, float, char, char[], or char*. All the usual formatting conventions are supported. Do **not** include the header file stdio.h in the rule file — CodeCheck uses its own internal declarations for the standard IO functions.

int sprintf(char * string, char * format, ...)

This function is similar to the ANSI stardard sprintf function. The argument variables may be of type int, float, char, or char*. All the usual formatting conventions are supported except the asterisk notion and the %n

format. Do not include the header file <stdio.h> in the rule file -- CodeCheck uses its own internal declarations for the standard IO functions.

int sscanf(char * string, char * format ...)

This function is similar to the ANSI standard sscanf function. The argument variables may be of type int, float, char or char*. All the usual formatting conventions are supported. Do not include the header file stdio.h in the rule file -- CodeCheck uses its own internal declaration for the standard IO functions.

void warn(int n, char * format, ...)

This function prints an error number and formatted message to the *stderr* stream, together with the filename and line number of the C source which triggered the message. If a listing file is open, then the warning message is echoed in the listing file with a marker indicating the position of the error. The warn function is modeled after the C function printf. The argument variables that follow the format string may be of type int, float, char, or char*. All formatting conventions are supported *except* the asterisk notation and the %n format.

Warning Messages

Warning messages from CodeCheck may originate from the evaluation of rules in a rule file, or they may originate from CodeCheck itself. In the former case the error number will carry the prefix **W**, while in the latter case the prefix is **C**. This is the only way to distinguish between these two kinds of warnings.

It is frequently extremely helpful to view warning messages in their complete context. To do this, use the **–L** command-line option. CodeCheck will then create a listing file, named check.lst. In this listing file every line is shown with its line number and error messages, with a marker showing the exact token that triggered each error message. Lines that were suppressed by the preprocessor (through #if conditionals) are shown with a hyphen substituted for the line number. If an error occurred within the expansion of a long or complicated macro, then use the **–M** option to show all macro expansions in the listing file. If an error occurred within a header file, then use the **–H** option to show all headers fully listed in the listing file.

Warnings Issued by Rules

Warning messages that originate from rule evaluation were written by the author of the rules, and are generated by the CodeCheck warn() function. These warning messages have the following format:

filename(line-number): Warning W*xxxx*: *text of message*

Macintosh version only: the second line of the message gives the complete pathname for the file in which the problem was encountered, and the line number that was being processed at the time the error was found, in the standard MPW format for error messages.

Error Warning Functions

User programmable Warning functions.

`char * err_message()`

Returns the message body of warning message numbered as CXXXX.

`int err_syntax`

Set to an integer when CodeCheck encounters a syntax error which is CXXXX. The value of the integer is 1 greater than value XXXX.

`fatal(n,str)`

Issue fatal error number with message string. CodeCheck will exit(-1) when this function is called.

`int warn( int num, char *control, char * message, … )`

Generates a warning message. First argument unique error number, following arguments similar to stdlib function printf(). This is an information function only, processing will continue.

Warnings Issued by CodeCheck

CodeCheck's own warning messages are informational only; they deal with situations that appear to be syntactic or semantic errors. These warning messages have a similar format (the only difference being the prefix on the error number).

filename(line-number): Warning C*xxxx*: *text of message*

After CodeCheck issues a warning error message, it attempts to proceed with further checking. Like any compiler, CodeCheck may become confused by a syntax error and issue a variety of nonsensical messages until finally encountering a fatal error condition. In this circumstance the only meaningful error message is the first.

C0000 Syntax Error.

A minor syntax error has been encountered, from which CodeCheck can usually make a graceful recovery.

CodeCheck may provide more information in an additional warning message, and will attempt to continue checking the source file.

C0001 Premature end of macro argument list.

There were fewer actual arguments in a macro call than there were formal parameters in the macro definition.

C0002 Missing macro argument.

An actual argument in a macro call was missing.

C0003 Too many macro arguments.

There were more actual arguments in a macro call than there were formal parameters in the macro definition.

C0004 Too many type modifiers.

The number of type modifiers (pointer to, array of, or function) in a declaration exceeded the maximum that CodeCheck can record.

The declaration should be simplified.

C0005 Empty #ifdef

An #ifdef or #ifndef directive was found without an argument.

CodeCheck assumes the value TRUE for the test.

C0006 Previous semicolon missing?

The syntax error on the indicated line may be due to a missing semicolon in a previous statement.

CodeCheck will attempt to continue checking, but all code between the missing semicolon and the marked semicolon will not be checked.

C0007 Missing right parenthesis.

CodeCheck expected a right parenthesis at the position marked in the listing, but did not find one.

CodeCheck will attempt to continue checking by pretending that there was a right parenthesis just before the marked position.

C0008 Macro defined differently in file <filename>

The differences between this macro definition and the definition in the given file are not trivial.

This is at best very poor style. All macro definitions should agree within a project.

C0009 Not a legal constant expression.

Standard C places many restrictions on what can appear in a constant expression. This expression violates at least one such restriction.

CodeCheck will attempt to continue checking by pretending that the constant expression was in fact legal.

C0010 String literal too long for CodeCheck.

The length of a string literal constant exceeded the capacity of CodeCheck's internal buffer. Note that ANSI C compilers are only required to handle string literals of length 509.

CodeCheck will truncate the string for purposes of further checking. This will limit its ability to find macros embedded in the string.

C0011 Type or storage class specifier required.

A declarator was found without a type or storage class specifier. This grammatical construction is obsolete, and should not be used.

C0012 Undeclared identifier <name>.

The specified identifier had not been previously declared.

Check the spelling of the variable. If it is correct, bring this message to the attention of Abraxas Technical Support.

C0013 Illegal or repeated typedef.

CodeCheck could not make sense of this type definition. Perhaps the identifier has already been defined as a type.

C0014 Error in member declaration.

A class, struct, or union member declaration has an error, possibly an undeclared type or illegal declarator.

C0015 Tag redefinition.

This enum, union, struct, or class tag has already been defined within the current scope.

C0016 <identifier> was declared differently in file <filename>.

A discrepancy was found between the current declaration and a previous declaration for the same identifier.

C0017 ANSI C prohibits type specifiers with typedef names.

The ANSI standard explicitly prohibits using type specifiers and typedef names within the same declaration.

CodeCheck will attempt to make sense out of the declaration, but agreement with nonstandard compilers cannot be guaranteed.

C0018 End-of-line found in a literal string or char.

A literal string or character constant was not properly terminated.

C0019 Rule triggers cannot have else-clauses.

The top-level *if*-statement in a CodeCheck rule has an *else*-clause. This is not permitted in CodeCheck rules, because the “else” condi-

tion describes an ill-defined event (how often does an event not occur?).

CodeCheck will ignore the else-clause. Rewrite the rule, if possible.

C0020 Error in declaration.

The indicated declaration contains a syntax error. One common reason: an undefined typedef name.

CodeCheck will ignore the troublesome declaration, and will attempt to continue parsing.

C0021 Floating-point constants found in a constant expressions.

A floating-point constant (a number with a decimal point or the F suffix) was encountered in a preprocessor constant expression.

CodeCheck will convert the constant to a *long* and continue. The result may not be what the programmer intended. **Note**: the ANSI C standard forbids floating-point constants in this context.

C0022 Tag type conflict for aggregate <tagname> in file <filename>.

An aggregated data structure (union, struct, or class) with the same tag name but a different tag type was defined earlier.

C0023 Could not open header file <filename>.

CodeCheck could not open the specified file.

Verify that the file exists, that it is not already open, and that it is in a directory known to CodeCheck (the **–I** command-line option can be used to specify directory paths).

C0024 Labels must be within function definitions.

A label was found outside any function definition.

C0025 Invalid argument for CodeCheck math function.

An inappropriate argument was passed to a mathematical function within a CodeCheck rule. For example, the *sqrt* function may have received a negative argument.

C0026 This function ought to be in the rule trigger.

Certain CodeCheck functions, *e.g. test_needed,* are designed to be used in the trigger of a rule. When used outside the trigger these functions may not work as intended.

C0027 This CodeCheck function has not been implemented.

(This error message should not occur.)

C0028 Invalid preprocessor constant expression.

A constant expression could not be evaluated by the preprocessor.

CodeCheck attempts to proceed using the part of the expression that has been evaluated up to this point. If the entire expression is empty, then CodeCheck assumes a value of zero.

C0029 String, comment, or character literal terminated by end of file.

An end-of-file marker was unexpectedly encountered while reading a string, comment, or character literal.

C0030 Divide by zero.

During evaluation of a CodeCheck rule a division by zero was attempted. ***The result was set to zero***.

C0031 This statement must go inside a rule.

During compilation of a rule file, CodeCheck found an executable statement that was not part of a rule. Every executable statement in a rule file must be embedded within a rule.

The rule programmer probably intended the statement to be executed either (a) once only, at the beginning of a project, or (b) at the beginning of the scan of each module. If (a) then embed the code within a rule that is triggered by prj_begin. If (b) then use mod_begin.

C0032 Nested comment.

A /* … */ comment was found embedded within a /* … */ comment. This will cause a syntax error unless the **–N** command-line option is in effect. Note that embedded comments are not permitted in ANSI C.

C0033 This trigger may describe an ill-defined event.

The CodeCheck rule compiler has reason to believe that the indicated rule will never be triggered. For example, the trigger event may be defined as a logical negation, such as:

```
if ( ! lin_has_comment )
    { ...
```

This event is not well defined because the set of all events that are not lines-with-comments is indeterminate. Rewrite such rules so that they are triggered by an event that positively will occur, *e.g.*

```
if ( lin_end )
    if ( ! lin_has_comment )
        { ...
```

C0034 Illegal parameter declaration.

This function declaration seems to mix old-style and new-style (prototyped) parameters. It is illegal in ANSI C to mix these styles.

C0035 A reserved keyword may have been used as an identifier.

Some older C compilers permit the use of certain ANSI and C++ reserved keywords (*e.g.* const and volatile) as identifiers. This syntax error may have been caused by such an identifier.

If you are using reserved keywords as identifiers, change them. Not only will this make CodeCheck happy, it will also greatly improve the maintainability and portability of your program.

C0036 Function definition expected here.

A function definition was expected but not found.

C0037 Nested class *tag1::tag2* cannot be found.

The specified class definition could not be found by CodeCheck.

If the class definition is in fact in the source code, and if your compiler has no trouble with the source code, then please fax a Trouble Report Form to Abraxas Technical Support.

C0038 Option <command-line option> not understood.

A command-line option did not make sense.

Please verify that the option you used is in the correct format. A brief explanation of all command-line options can be obtained by invoking CodeCheck with no arguments.

C0039 Bad message.

This return code should never occur.

C0040 Improper call to CodeCheck function *new_type()*.

The function new_type() was used incorrectly in a CodeCheck rule. The actual text of this warning will specify exactly what was wrong with the call.

C0041 Do not declare CodeCheck variables with initializers.

An initializer was found in a declaration in a CodeCheck rule file.

Remove the initializer, and set the variable appropriately within a rule that is triggered by mod_begin.

C0042 CodeCheck was confused by this C++ class initializer.

This message should never occur. Please fax a copy of the code that caused this message to Abraxas Technical Support.

C0043 Trigraph in character literal: replace ??' with ?\?'.

Two question marks followed by a single quote is an ANSI trigraph, a symbol sequence that is interpreted by ANSI compilers as the carat ^ symbol (which is not present on some European keyboards). This trigraph was found within a character literal, implying that it was not intended to be a trigraph.

Insert a backslash character between the two question marks. This will prevent syntax errors when your code is parsed by ANSI compilers, and will not change the meaning of the code under any compiler.

C0044 Do not use template arguments with a constructor name.

According to the Annotated C++ Reference Manual, page 350, it is not syntactically correct to use template arguments in the declaration of a constructor or destructor for a template class.

CodeCheck will ignore the template arguments. The arguments should be removed, as they will make the template definition nonportable.

C0045 Member <name> not found within <tag>.

The named member could not be found within the union, struct, or class.

C0046 Attempt to modify a constant.

An attempt was made either to assign a value to a constant, or to increment or decrement a constant using the ++ or –– operator.

C0047 Assignment incompatibility.

An assignment could not be compiled because the destination type is not compatible with the source type.

C0048 Function return type incompatibility.

The type of the value in a return statement is not compatible with the declared return type of the function.

C0049 Argument <k> incompatible with prototype.

The type of the k^{th} argument in a function call is not compatible with the declared type of the corresponding formal parameter in the function prototype currently in scope.

C0050 There is no class to inherit from.

The inherited keyword has been used when there is no class to inherit from (Symantec THINK C for Macintosh only).

C0051 Template <name> has not yet been declared.

A C++ template name has been used before it is declared. Several C++ compilers consider this to be legal, but it is very poor programming style.

Fatal Error Messages

Fatal error messages from CodeCheck may originate from the evaluation of rules in a rule file, or they may originate from CodeCheck itself. In the former case the error number will carry the prefix **F**, while in the latter case the prefix is **E**. This is the only way to distinguish between these two kinds of warnings.

CodeCheck's own fatal error messages indicate a very severe problem, one that prevents CodeCheck from processing your program any further. These messages have the following format:

filename(lineno): Fatal Error E*xxxx*: fatal error message

After CodeCheck displays a fatal error message, it terminates without any further checking.

E0000 Syntax Error

CodeCheck encountered a syntax irregularity from which it could not recover. The actual text of the error message may provide some detail on what was found. If any syntax warnings preceded this fatal error, then the actual problem may have occurred earlier.

First make sure that your source code really does compile without error on your C compiler. Second, examine the source code in the lines preceding the error message for any unusual constructions that may be peculiar to your compiler. Third, follow the suggestions in the section entitlted "Trouble-shooting".

E0001 CodeCheck is out of dynamic memory.

CodeCheck ran out of dynamic memory space. This usually means that this source file has too many macro definitions, type definitions, and variable names for CodeCheck to keep track of.

MS-DOS: Are you using the version of CodeCheck that makes use of your extended memory? If so, then add more extended memory. If not, then try the extended memory version.

Macintosh: Increase the memory allocated by the system to the MPW Shell (the amount is found in the "Get Info" box for the Shell).

If all else fails, try using the **–Z** command-line option to inhibit cross-module checking. This greatly reduces the demands made by Code-Check on dynamic memory.

E0002 "filename" or <filename> expected but not found.

An #include preprocessor directive was found without a filename specified in the proper format.

Make sure that the filename is enclosed in quotes or angle brackets.

E0003 Macro name expected.

A #define preprocessor directive was found without a valid identifier for the name of the macro.

Make sure that the macro has a name that begins with an alphabetic character.

E0004 Unexpected end of macro definition.

The end of a macro definition was encountered, without a close parenthesis for the macro formal parameter list.

Verify that the macro definition is syntactically correct.

E0005 Invalid macro formal parameter.

A formal parameter in a macro definition was not a valid identifier.

Verify that each formal parameter is an alphanumeric string that starts with an alphabetic character. Formal macro parameters must not be expressions, and they must not be missing.

E0006 Comma expected in macro argument list.

Two formal parameters in a macro definition were separated by something other than a comma.

Check the formal parameter list.

E0007 Macro arguments found but not expected.

There were actual arguments in a macro call, but no formal parameters were given in its definition.

Correct the macro call.

E0008 Too many files.

CodeCheck's internal array of filenames has overflowed.

Please report this condition to Abraxas Technical Support.

E0009 Multiple #else **directives.**

More than one #else preprocessor directive was encountered after an #if, #ifdef, or #ifndef.

Check for improperly balanced #if - #else - #endif directives.

E0010 Dangling #else **directive.**

An #else preprocessor directive was encountered where none was expected.

Check for an #else directive that should have been deleted.

E0011 Dangling #endif **directive.**

An #endif preprocessor directive was encountered where none was expected.

Check for an #endif directive that should have been deleted.

E0012 Overflow in paste buffer.

The length of tokens being pasted together with the ANSI paste operator (##) has exceeded the maximum permitted by CodeCheck.

Verify that the specified paste operation is correct. If it is, then try rewriting the code in a simpler fashion.

E0013 Buffer overflow in ungetch**.**

A CodeCheck internal buffer has overflowed.

Please report this condition to Abraxas Technical Support, with a copy of the listing file from CodeCheck (if possible).

E0014 Exponent expected but not found.

During processing of a floating-point constant, the letter 'e' or 'E' was encountered with no following exponent.

Determine the correct exponent and place it after the 'e' or 'E'.

E0015 Not allowed in a CodeCheck rule.

A construction was found within a CodeCheck rule that is valid in C or C++ but not in CodeCheck rules.

Review the differences between the CodeCheck rule grammar and the grammar for C. Remember that CodeCheck rules use a restricted subset of the C language.

E0016 Pseudocode buffer overflow.

An internal CodeCheck buffer overflowed during compilation of a rule file.

Please report this condition to Abraxas Technical Support, with a copy of the rule file from CodeCheck (if possible).

E0017 Undeclared identifier <name>.

A CodeCheck variable had not been previously declared.

Check the spelling of the variable. If it is correct, remember that *all* CodeCheck variables must be declared.

E0018 Too many triggers for this rule.

An internal CodeCheck buffer overflowed during compilation of a rule.

Please report this condition to Abraxas Technical Support, with a copy of the rule (if possible). Try to simplify the rule, using fewer variables.

E0019 Nesting of if-else statements exceeds maximum.

An internal CodeCheck buffer overflowed during compilation of a rule.

Please report this condition to Abraxas Technical Support, with a copy of the rule (if possible). Try to simplify the rule, using fewer if-statements.

E0020 Duplicate declaration.

Two variables or typedef names have been declared with the same name, at the same level.

Check the spelling of both names. If correct, then check for an erroneous redeclaration.

E0021 CodeCheck name table overflow.

Too many user-declared CodeCheck variables have been declared in a rule file.

Please report this condition to Abraxas Technical Support, with a copy of the rule file (if possible). Try to simplify the rule file, using fewer user-declared variables.

E0022 NULL format string in warn()**,** printf()**, or** fatal()**.**

E0023 CodeCheck integer storage has been exceeded.

Too many integers have been used in a CodeCheck rule file.

Simplify your rules, and *contact Abraxas Technical Support.*

E0024 CodeCheck float storage has been exceeded.

Too many floating numbers have been used in a CodeCheck rule file.

Simplify your rules, and *contact Abraxas Technical Support.*

E0025 This type specifier is not allowed in a CodeCheck rule.

E0026 Too many parentheses in declaration.

Too many parentheses were used in the construction of a declarator.

Simplify this declaration.

E0027 Value stack overflow.

The nesting level of a declaration, expression, or statement exceeded that allowed by CodeCheck.

Please inform Abraxas Technical Support if this condition occurs.

E0028 Underflow of value stack.

This condition should never occur.

E0029 Improper #elif **syntax.**

Possibly an #else was used instead of an #elif before this line.

Check for proper balancing of #if - #elif - #endif directives.

E0030 Dangling #elif **directive.**

An #elif was encountered without a prior #if, #ifdef, or #ifndef.

Check for a missing #if directive.

E0031 Invalid constant expression.

An error was found in a constant expression (an expression that must be evaluated at compile-time, not execution-time).

Check for the use of C operators or functions (*e.g.* sizeof) that are not permitted within constant expressions.

E0032 Float type is not allowed in preprocessor arithmetic.

A constant of type float was found in a constant expression.

C does not permit floating arithmetic within constant expressions.

E0033 <identifier> is NOT a statistic!

A non-statistical CodeCheck variable was used within a rule as though it were a statistic.

Only certain predefined CodeCheck variables are statistics. Non-statistical predefined variables cannot be placed in the statistic storage class by the user.

E0034 Too many cases for the statistic <identifier>.

More than 100,000 cases were recorded for a statistical variable.

Please contact Abraxas Technical Support if you require more cases.

E0035 Strings are not allowed here.

A string was used inappropriately within a CodeCheck rule.

E0036 Too many predefined statistic variables!

There is a limit to the number of statistical variables that can be active within one rule file.

Try to reduce your usage of statistical variables, or break up the rule file into several components.

E0037 printf**: unsupported type.**

An attempt was made to print a variable of a type not supported by the CodeCheck printf function.

E0038 Do not modify predefined CodeCheck variables!

An attempt was made to assign a value to a predefined CodeCheck variable.

E0039 Attempt to reset a non-statistical variable: <identifier>

The reset() function cannot be used on non-statistical variables.

E0040 Not an lvalue!

The expression on the left-hand-side of an assignment operator within a CodeCheck rule is not valid.

E0041 Do not use ++ or -- on statistical variables!

The pre- and post- increment and decrement operators may not be used on CodeCheck variables of the statistic storage class.

E0042 Correlation requires equal numbers of cases!

The *corr* function in a CodeCheck rule was called with two arguments that have unequal numbers of cases.

A correlation between two variables can only be calculated when the two variables have the same numbers of cases.

E0043 Not a legal declaration.

A declaration was expected. What was found was apparently not a declaration.

E0044 Invalid stringize operand.

The ANSI stringize operator (#) was applied to an invalid (or null) operand.

The operand must be a formal parameter of the macro.

E0045 Illegal function declarator.

A function declarator was expected but not found.

E0046 Constant buffer overflow.

This CodeCheck rule file uses more constants than the compiler has space for.

Please contact Abraxas Technical Support if this happens.

E0047 Input line exceeds buffer length.

An input line for a C source file or a CodeCheck rule file was longer than 2050 characters.

E0048 Variable <identifier> has not yet been implemented.

The specified variable will be implemented in a later release.

Check with Abraxas Software to see if you have the latest release.

E0049 <type identifier> does not make sense here.

Among the common causes for this error are: (1) a header file containing a necessary type definition was not included, or (2) a needed type definition is hidden within conditional code. *In either case the usual root cause of the problem is that a macro symbol is undefined*. Possible reasons: (1) the code assumes that the macro symbol will be defined in the command-line, or (2) the macro symbol is a non-ANSI symbol that is predefined in your C compiler.

Create a listing file using the –H and –M command-line options, and search through the listing for the definition of the typedef name that was involved in the declaration that caused this syntax error. Check to see whether the definition was suppressed by the preprocessor (it was suppressed if its line number is a hyphen), or if the definition is in a header file that was never included.

E0050 Illegal declaration.

This message is usually a side-effect of an earlier syntax warning.

E0051 Internal CodeCheck error.

Please report this message to Abraxas Technical Support.

E0052 Unknown preprocessor directive.

A nonstandard preprocessor directive was encountered that is not known to CodeCheck. *Please send documentation on the meaning of this directive to Abraxas Technical Support.*

E0053 Missing right curly brace?

The close brace of a function definition was expected here but not found. If your code looks correct then please fax a Trouble Report Form to Abraxas Technical Support.

E0054 Too many directories.

Too many full path names were used during project checking (the default is 64 in CodeCheck version 5).

E0055 CodeCheck aborted by user.

The user hit the control/C key on the keyboard.

E0056 #error **directive encountered.**

The preprocessor encountered an #error directive, which caused checking to terminate.

These directives are placed in programs to prevent compilation when certain necessary conditions are not met. To determine why the occurred, run CodeCheck again with the **–H** option and study the listing file (check.lst) that is produced. Usually a macro has the wrong value, or is not defined when it should be defined. Determine what the correct value of this macro should be, and define or undefine it on the command-line with a **–D** or **–U** option.

E0057 Allowed in C++ but not in C.

The indicated syntax is legal C++, but not C. Make sure that the correct –K command-line option has been used for this source code.

E0058 NULL string argument in CodeCheck strcmp **function.**

One of the arguments to strcmp was NULL.

E0059 Paste operator (##) is the first token.

The ANSI preprocessor paste operator (##) cannot be the first token in a macro expansion. This is a syntax error in a preprocessor macro definition or macro expansion.

E0060 CodeCheck will not write to any file with extension <extension>.

As an elementary security feature, the CodeCheck fopen() function will refuse to open a file for writing if its extension is one of the following: .c, .cp, .cpp, .h, .hpp.

Limits and Assumptions

Like all compilers, CodeCheck has certain built-in limits and assumptions. These are summarized below. Note that some operating systems (notably MS-DOS) impose severe memory restrictions not reflected in the summary below.

Cases per switch	Not limited.
Characters per line	Not limited, but only 255 are displayed.
Constants per *enum*	Not limited.
Depth of *#ifdef* nesting	Not limited.
Depth of ***#****include* nesting	Not limited.
Depth of parenthesis nesting	Not limited.
Depth of *struct* nesting	Not limited.
Function formal parameters	Not limited.
Length of file names	256 characters.
Length of path names	512 characters.
Length of identifiers	First 75 characters are significant.
Length of macros	Not limited.
Length of string literals (after concatenation)	1000 characters (MS-DOS) 5000 characters (all other operating systems)
Macro formal parameters	31.
Members per *struct* or *union*	Not limited.
Number of global declarators	Not limited.
Number of local declarators	Not limited.
Number of file names	A maximum of 4000 distinct file names can be used in a project. Contact Abraxas if this is not sufficient.

Number of lines per file	Not limited.
Number of macros	Not limited.
Number of type modifiers	A maximum of 12 type modifiers (*array of…*, *pointer to…*, *reference to…*, or *function returning…*) can be attached to any one declarator.
Preprocessor character set	The CodeCheck preprocessor and compiler use the same character set. Preprocessor characters can be negative.
Signedness of type *char*	CodeCheck assumes that char is signed.
Size of type *int*	CodeCheck assumes that an int is larger than a short, and shorter than a long.

Trouble-shooting Techniques

It sometimes happens that code which compiles without error on your C or C++ compiler will generate a syntax warning or fatal error when scanned by CodeCheck. The most likely causes for these errors are:

- A macro that is required by your system or library header files is not defined.
- You specified the wrong **–K** option, so CodeCheck failed to recognize a special keyword or macro, or interpreted an identifier as a keyword.
- Your compiler has one or more nonstandard keywords that are not known to CodeCheck. You can define new base types with the function new_type().

First run CodeCheck again on the same source file, but specify command-line options –**H** and –**M**, and do not use –**J**. This will create a listing file named check.lst with all headers listed, all macros expanded, and all warnings shown in context. Open this listing file and search for the first warning message.

It is important to search for the *first warning message*. Sometimes, when a fatal error occurs, there may be one or more warning messages that are numbered as Cxxxx preceding the fatal error message. Since CodeCheck parser can tolerate some syntax error at certain extent, after the first warning occurs, CodeCheck try's to proceed the parsing on the source code. In the case of a syntax error, CodeCheck will have misinformation or misinterpretation in following the source code. When the error become irrecoverable. A fatal error is generated. Quite often, after first warning message is removed, the fatal error also will disappear.

It very commonly happens that your system and library header files have conditional code that either ought to have been suppressed by the preprocessor, or ought not to have been suppressed. You can tell when the CodeCheck preprocessor has suppressed code by looking at the line numbers in the left-hand side of the page. When code is suppressed the line number is absent. Examine all the code that precedes the first warning message to see if it was suppressed when it ought to have been, or *vice versa*. This process can be very educational: you may find conditional code for features that you never knew existed. If you discover that a macro should have been defined (or undefined), then run CodeCheck again with the appropriate –**D** and –**U** options.

If the error seems to be associated with a common nonstandard keyword (*e.g.* near, far, huge, cdecl, pascal, interrupt) that should have been recognized by CodeCheck, then it is likely that you specified –**K0** or –**K1** instead of –**K2** or –**K3**. Remember that strict ANSI C does *not* include these keywords.

If the error seems to be associated with an unusual keyword (*e.g.* packed) or an unusual grammatical construction, then it is likely that your compiler has some special features that Abraxas would like to know about. Let us know all the details, preferably by fax. Meanwhile, if it looks as though the code would be grammatical if CodeCheck were to ignore the special keyword, then a work-around may be possible. For example, users of the Microtec C compiler should always insert this rule into their custom rule files:

```
if ( mod_begin )
   {
   ignore( "packed" );
   ignore( "unpacked" );
   ignore( "interrupt" );
   }
```

This rule will cause the CodeCheck lexical analyzer to skip over every occurrence of packed, unpacked, and interrupt. Try checking your code again with a rule like this. If it now parses without error then you have found a solution. As another example, this rule file will prevent syntax errors for users writing for the Symantec C compiler (these are *not* needed for Symantec C++):

```
if ( mod_begin )
   {
   ignore( "__handle" );
   undefine( "_MSC_VER" );
   define( "__ZTC__", "0x0300" );
   define( "asm", "_intrinsic_" );
   }
```

It may also be possible to use a macro defined with the –**D** option to eliminate this kind of error. For example, the command

```
check -K0 –Dvoid=int foo.c
```

will invoke CodeCheck with the K&R keyword set, and the non-K&R keyword void defined as a macro with the value int.

Trouble Report Form

Fax to: **Abraxas Technical Support**
Fax number: **+ 503-232-0543**

From: _____

Company: _____

CodeCheck Version: _____ Abraxas Part Number: _____

Operating System: _____ Platform: _____

C or C++ compiler: _____

Your phone number: _____

Your fax number: _____

Please describe your problem. *If it is a syntax warning or fatal error, please read the section on Troubleshooting and try the suggestions found there before faxing in a Trouble Report Form!* It frequently helps to show us the relevant portion of a listing file, so that we can see the error message in its exact context. Make this listing file by running CodeCheck with the **–H** and **–M** options. Do not use **–J**.

Index