

# ***C and C++*** **Source Code Analysis** ***using*** **CodeCheck**

*by*  
*Loren Cobb, PhD.*

CodeCheck™ is a product of Abraxas Software, Inc.  
CodeCheck was designed & written by Loren Cobb.

*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](mailto:support@abxsoft.com)*

*[www.abraxas-software.com](http://www.abraxas-software.com)*

# Table of Contents

<b>PREFACE.....</b>	<b>V</b>
<b>ACKNOWLEDGMENTS.....</b>	<b>VI</b>
<b>QUICK START:</b>	
<b>0.1 INSTALLATION.....</b>	<b>1</b>
<b>0.2 COMMAND-LINE OPTIONS.....</b>	<b>2</b>
<b>0.3 CODECHECK FILE NAMES .....</b>	<b>9</b>
<b>0.4 HOW TO USE CODECHECK.....</b>	<b>10</b>
<b>INTRODUCTION TO CODECHECK:</b>	
<b>1.1 THE ELEMENTS OF STYLE.....</b>	<b>13</b>
<b>1.2 WHY PROGRAMS BREAK.....</b>	<b>15</b>
<b>1.3 WHY PROGRAMS FAIL TO PORT .....</b>	<b>18</b>
<b>1.4 THE STRUCTURE OF CODECHECK.....</b>	<b>21</b>
<b>1.5 DEBUGGING WITH CODECHECK.....</b>	<b>23</b>
<b>1.6 PREDEFINED MACRO CONSTANTS .....</b>	<b>24</b>
<b>CHECKING TYPES:</b>	
<b>3.1 HOW TO ANALYZE A TYPE DECLARATION.....</b>	<b>28</b>
<b>3.2 HOW TO DETERMINE THE TYPE OF AN IDENTIFIER.....</b>	<b>31</b>
<b>3.3 HOW TO DETERMINE THE TYPE OF AN OPERAND.....</b>	<b>31</b>
<b>3.4 HOW TO DETECT IMPLICIT TYPE CONVERSIONS .....</b>	<b>32</b>
<b>PORTABLE STYLE:</b>	

4.1	LEXICAL ISSUES IN PORTABILITY.....	34
4.2	PREPROCESSOR CONSIDERATIONS .....	46
4.3	PORTABILITY IN DECLARATIONS .....	56
4.4	PORTABILITY AT THE EXPRESSION LEVEL .....	64
4.5	PORTABILITY OF FUNCTIONS.....	68
4.6	C COMPILER LIMITS .....	70
 MAINTAINABLE STYLE:		
5.1	LEXICAL ISSUES IN PROGRAM MAINTENANCE.....	74
5.2	PREPROCESSOR CONSIDERATIONS .....	92
5.3	MAINTAINABILITY IN DECLARATIONS.....	95
5.4	MAINTAINABILITY AT THE PROJECT LEVEL .....	100
 SOFTWARE METRICS:		
6.1	PROGRAM SIZE.....	101
6.2	LOGICAL COMPLEXITY .....	115
6.3	CODE DENSITY .....	121
 CODECHECK RULE SETS:		
7.1	VERIFYING POSIX.1 COMPLIANCE.....	129
7.2	COMPLIANCE WITH CODING STANDARDS.....	136
7.3	PORTING TO ANSI C.....	140
7.4	PORTING TO STRICT K&R COMPILERS .....	147
7.5	MEASURING CODE COMPLEXITY.....	149
7.6	VERIFYING THE ORDER OF MODULE ELEMENTS .....	151

**7.7 C++ RULES ..... 153**

**7.8 ADVANCED C++ RULES ..... 156**

**SUPPORTING MATERIAL:**

**8.1 GLOSSARY..... 171**

**8.2 BIBLIOGRAPHY..... 178**

**8.3 INDEX..... 181**

# Preface

Producing accurate, reliable and flexible programs in C or C++ is a difficult task. Even experienced programmers need tools to aid in the program development process, but all too few tools exist in today's market that can detect bugs in C and C++ source code and help the programmer to avoid problems.

CodeCheck is a powerful tool for analyzing C and C++ source code. Unlike other tools, Codecheck is itself fully programmable. 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.

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 to compliance with style standards.

# Acknowledgments

We gratefully acknowledge the invaluable help given to the CodeCheck project by the following individuals, who contributed suggestions and bug reports. We couldn't have done it without you!

Jan-Anders Åkerholm  
Wendy Averdung  
George Baker  
Wahab Baldwin  
Ed Batutis  
Nasser Bazzi  
John Benson  
Bill Bentley  
Dana Birkby  
Dale Bremer  
John Bradley  
Mike Branson  
Linda Brigham  
Jeff Brown  
Van Brollini  
Thomas Brustbauer  
Walt Buehring  
Laura Burke  
Bill Campbell  
Pat Cappelare  
Camille Carum  
Rob Chambers  
Alex Chervet  
Tim Child  
John Clinton  
Patrick Conley  
Darryl Cornish  
Bill Costello  
Kevin Coyle  
Mike Curry  
Mark De May  
Matt Diamond  
David Doerner  
John Doggett  
Bob Domitz  
Tom Dropka  
George Entwistle  
Richard Evans  
Aaron Fager  
Brent Fairbanks  
Bud Feuleless  
Steve Fine  
Julianne Fontenoy  
Keith Fulton  
Greg Germano  
Bud Feuleless

Keith Fulton  
Shawn Garbett  
Jerry Garcia  
Bonnie Gilmore  
Dennis Glenn  
Soo Hye Goh  
David Gordon  
Bruce Graham  
Elaine Granoff  
Jeff Johnson  
Brett Halle  
Esko Hannula  
Othar Hansson  
Tris Harkless  
Bill Hazzard  
John Herbold  
Alison Hine  
Paul Hurley  
Jim Jacobson  
David Johnson  
Mike Johnson  
Darrell Jones  
Arun Joshi  
Ken Joyner  
Ed Kirk  
Ian Koenig  
Tom Kohler  
Hannu Kokko  
Detlef Kowalewski  
Ron Kuhn  
Patrica Langer  
Mark Lamer  
Eric Lear  
David Linsky  
Alan Liu  
Martin Lord  
Tom Lucas  
Frank Lusardi  
Paul McGlashan  
Bill McMahon  
Terry McNulty  
Eric Melbardis  
Mike Muegel  
Marcel Meyer  
Deborah Miller  
Steve Monett

Stephen Montgomery  
Tom Moreaux  
Peter Morse  
Mike Muegel  
Greg Munger  
Rick Murnane  
Hugh Njemanze  
John Norby  
Michael O'Leary  
Ingmar Olsson  
Lyle Parkyn  
Bob Peterson  
Steve Peterson  
Greg Pilkington  
Darrel Pinson  
Th. Pfister  
Karl Pingle  
John Plocher  
Alan Pope  
Chris Prendergast  
Steve Ray  
Mike Reid  
Steve Reynolds  
Dan Richards  
Robin Riley  
Kay Roche  
Jim Roskind  
Stefan Roth  
Florian Sachse  
Richard Sargent  
Jay Sarkar  
Alan Sauls  
Karl Schopmeyer  
Rick Schuessler  
Peter Schwaller  
Roshin Sharma  
Andrew Shebanow  
Ursula Shelander  
Hartmut Stein  
Ursula Shelander  
Jeffrey Smith  
Cass Smith  
Tim Southgate  
Brian Stromquist  
Dan Sullivan  
Malcolm Sutter

Padma Talasila  
Larry Thiel  
Julie Tiemann  
Esther Tong  
Gino van den Bergen  
Thomas Wikshult  
Clayton Wilkinson  
David Williams  
Roderick Williams  
Tim Wint  
Aaron Wohl  
Matt Woodward  
Heinz Wrosch  
Cornelia Yoder  
Robert Yu  
Doug Zimmerman  
Bruce Zimov

# Chapter 0: Quick Start

## 0.1 Installation

1. Copy the CodeCheck program into the directory in which you normally keep your programming tools.
2. Create a new directory for the collection of CodeCheck rule files that is supplied on the distribution disk. Copy these rule files to this new directory.
3. Assign the pathname of the rule directory that you created in Step 2 to an environmental variable named `CCRULES`. CodeCheck will use this variable to locate rule files. Multiple pathnames may be specified in this environmental variable if this is desired. Separate each pathname with the character normally used in your operating system (Unix: **colon**, DOS & OS/2: **semicolon**, Macintosh: **comma**). On Unix systems only, if this variable is not defined then CodeCheck will look for rule files in a directory named `/usr/CodeCheck`.
4. CodeCheck looks for header files in the paths listed in the `INCLUDE` environmental variable. (On Macintosh systems it looks for `CIncludes`.) Make sure that this variable is correctly defined. *This is important: CodeCheck cannot function without access to the same header files that your C or C++ compiler uses.* Multiple pathnames may be specified in this environmental variable if necessary.
5. If you have header files (e.g. system headers) that you wish CodeCheck to read but not check, then assign the pathnames of the directories containing these headers to an environmental variable named `CCEXCLUDE`. This step is not necessary for CodeCheck: it is useful only when you wish to apply rules to some but not all of the header files that are included in your C or C++ source files.

## 0.2 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 (*excluding* the names of header files and libraries). Give this project file the extension `.ccp`. Then invoke CodeCheck, specifying the project file instead of a source file:

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

CodeCheck will apply its rules to each source file named in `myproject.ccp`, and will apply project-level checking across all the files in the project. The `ccp` extension informs CodeCheck that the specified file is a project file rather than a C source file. This extension may be omitted in the command-line. 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.

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.

- 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.
- 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.
- Kn** 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++
  - 5 Symantec C++
  - 6 Borland C++
  - 7 Microsoft C++
  - 8 IBM Visual Age C++
  - 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 recompile 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.

- Sn** 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 <ctype.h> // A standard system header
#include "project.h" // An application header
```

When option **-S1** is in effect, CodeCheck will apply its rules to *project.h* but not *ctype.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 prototype file will be created in the current directory, unless a target directory is specified with the **-Q** option.
- U** Undefine a macro constant. The name of the macro must follow immediately. Thus `check -UMSDOS 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 of the Reference Manual for usage suggestions.
- W** For CodeCheck users. See Section 1.4 of the Reference Manual for usage suggestions.
- X** For CodeCheck users. See Section 1.4 of the Reference Manual for usage suggestions.

- Y For CodeCheck users. See Section 1.4 of the Reference Manual 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.

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 Sections 1.3–1.5 of the Reference Manual for details). Option **-X** is recommended for users who wish to design custom rule files whose behavior is controlled by a command-line option.

## 0.3 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 prefix.
- .ccp** A project file for CodeCheck. This file contains a simple list of the filenames of all of the source modules that comprise a project, one filename per line. Header files and libraries should not be listed in this file.

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

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

## 0.4 How to Use CodeCheck

### 0.4.1 A Single User with Prepackaged Rules

Let us suppose that you simply want to check your C source file `foo.c` for some of the common errors that are not usually detected by C compilers. You want to see the warning messages in context, in a listing file. The command is

```
check -Rerror -L foo.c
```

When CodeCheck completes execution, open the listing file `check.lst` with your editor. Each warning will be shown under the line that caused the warning, with a marker immediately under the token that was being scanned when the error was detected.

This command made use of the prepackaged rule file `error.cc`, supplied by Abraxas. Some other prepackaged rule files that you may find helpful are:

*Tutorial and example rule files:*

- `dcl.cc.cc` Example rules that use the declarator variables.
- `cplist.cc` Lists and describes all classes in each module.
- `cplus.cc` Example rules for C++ style checking.
- `declare.cc` Interpret global declarations in ordinary “English”.
- `fnccalls.cc` Generate a list of functions called by each function.
- `forward.cc` Example rules that illustrate forward chaining.
- `lex.cc` Example rules that use the lexical variables.
- `nesting.cc` Example rules for measuring iteration nesting.
- `oometric.cc` Computes several object-oriented metrics.
- `order.cc` Check for standard ordering of file elements.
- `pp.cc` Example rules that use the preprocessor variables.
- `prefix.cc` Example rules for checking declarator prefixes.
- `sample.cc` Example rules for compliance with standards.
- `wrapper.cc` Detect headers and `#includes` that are not “wrapped”.

*Production rule files:*

- `ansi.cc` Check for compatibility with the ANSI C standard.
- `BSD43.cc` Check for use of BSD 4.3 features that are not POSIX.
- `braces.cc` Check for consistent use of braces in high-level statements.

- `complex.cc` Measures of program complexity (McCabe, etc).
- `error.cc` Check for errors that compilers may not find.
- `fromHP.cc` Check for portability from HP/Apollo C.
- `fromVAX.cc` Check for portability from VAX C.
- `general.cc` Check for general portability.
- `indent.cc` Check for proper indentation.
- `Halstead.cc` Measures of program size developed by Halstead.
- `logical.cc` Check for if-conditions that are too complex.
- `maintain.cc` Check for general maintainability.
- `posix.cc` Check for violations of the POSIX namespaces.
- `size.cc` Measures of program size based on lines & statements.
- `style.cc` Check for compliance with Comeau's C style standards.
- `SVID.cc` Check for use of SVID features that are not POSIX.
- `toIntel.cc` Check for portability to the Intel iC-386 V4.2 compiler.
- `toKR.cc` Check for portability to the 1978 K&R C standard.
- `toMPW.cc` Check for portability to Macintosh MPW C version 3.2.
- `toSuncpp.cc` Check for portability to Sun C++ version 2.1.
- `toVAX.cc` Check for portability to VAX C.

#### **0.4.2 Multiple Users with Custom Rules**

Large corporations with many programmers often have staff assigned to maintaining the tools used by these programmers, including tools like CodeCheck which are used for quality assurance. It is often appropriate to assign to a single individual the responsibility to write and maintain a CodeCheck rule file that encodes the corporate standards for C style. This compiled rule file would be placed in a network directory with the name `default.cco`. Then each programmer can check his or her code with a command like:

```
check foo.c
```

Assuming that each programmer has defined an environmental variable `CCRULES` that points to the directory containing `default.cco`, this command will cause CodeCheck to apply the corporate rules to his or her source code.

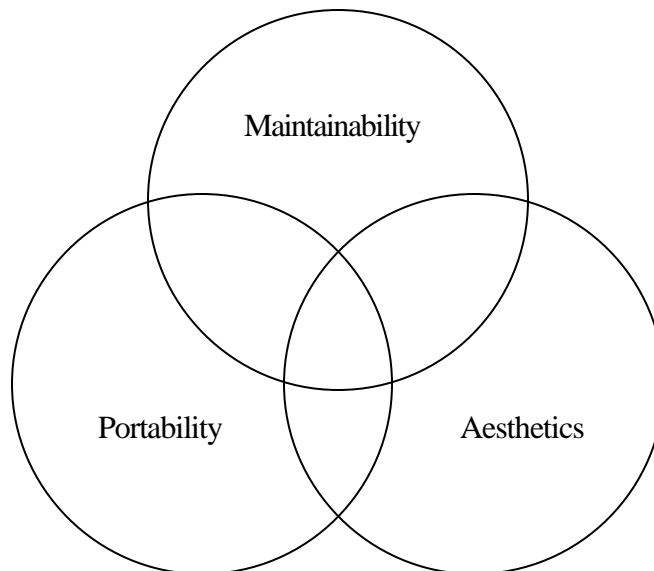
*Note:* Please contact Abraxas for site license information.

# Introduction to CodeCheck

## 1.1 The Elements of Style

Every programmer has a distinctive style of writing. This style is an expression of many things: the programmer's sense of æsthetics, the demands of speed and efficiency, the requirements of the customer, the needs of maintenance programmers, and the possibility that the program will need to be ported to another computer or translated into another language. Many of these elements of style require careful value judgments by the programmer or project leader. Once the stylistic requirements are clearly defined, CodeCheck can be an invaluable tool for monitoring each of these elements of program style.

The principal elements of good programming style are the requirements of æsthetics, maintenance, and portability. Fortunately, there are significant overlaps between these elements, as illustrated below. Can C programmers achieve a style that is at once portable, easily maintainable, and elegant? The answer is an emphatic "yes", and CodeCheck can help any C or C++ programmer to develop his or her personal style towards this goal.



**Figure 1: Overlap among the elements of good programming style.**

There are several general principles that govern the overlap between these elements:

1. ***Code that is technically “portable” but not easily maintained is not truly portable.*** A nontrivial C program that is universally portable is a very rare animal indeed. There are so many dark and dusty corners in the syntax and semantics of C that universal portability is next to impossible. Therefore, programmers must make the purpose of their portable code evident — and this is the essence of maintainability.

2. ***Code that is elegant without being maintainable will have a short life.*** What good is elegant code if even its author cannot understand it one year later? The C grammar lends itself to code that is highly abstract and superficially elegant, especially in the area of character processing, but maintenance programmers may consider this style to be anything but aesthetically pleasing. What is wrong here is the equation of elegance with aesthetics: the two concepts are not identical.

3. ***Simplicity lies at the heart of portability, maintainability, and aesthetics.*** For reasons difficult to understand, many C programmers never learn this essential truth. Quite possibly the fault lies not in the language itself, but in the culture that has grown up around it. This culture seems to value complexity, density and abstractness over all other considerations, perhaps for the sheer fun of creating puzzles that others find impossible to solve. Whatever the reason, these values militate against clean, simple and understandable code.

## 1.2 Why Programs Break

Despite the C language's reputation for portability, it is an unfortunate fact of life that an apparently flawless C program will usually fail when compiled with a different compiler, or under a different operating system, or on another type of computer. The reasons for this fact of life are many, but there are three principal forces at work which underlie almost all such problems.

### 1.2.1 Force #1: Language Parochialism

First, most programmers become thoroughly versed in only one implementation of C, on only one computer, under only one operating system. As they learn more about this programming environment, they unwittingly begin to use its many nonstandard features, and to take advantage of each of its quirks and foibles. These nonstandard features are typically concentrated in the lowest levels of operation: the preprocessor and the lexical analyzer of the compiler, the file management routines of the operating system, and the bit-manipulation instructions of the machine. Inexperienced programmers do not realize the extent to which computer languages are dependent on low-level conventions, and are wholly unaware of the implications of using nonstandard features. Unfortunately, these low-level nonstandard features tend to spawn an unending stream of the most amazingly mysterious bugs, often months or years after a program is first written.

### 1.2.2 Force #2: Programmer Machismo

The second force at work to make programs break is programmer machismo. Many programmers are young, smart, brash, fearless, and anxious to prove themselves to be wickedly clever. These are the programmers who write macros like

```
#define put(x,p) (--(p)->cnt>=0?(*(p)->ptr++=(x)):flush(x,p))
```

and think that by getting all this power on one single completely undocumented line they have achieved something special. What they have actually created is a maintenance nightmare for someone else. (This delightful example is due to Andrew Koenig, who dissects a slightly different version on page 80 of **C Traps and Pitfalls**).

### 1.2.3 Force #3: Compiler Drift

The third program-destroying force operates not on application programmers, but on compiler writers. This force is the almost irresistible temptation to include a new feature or language extension (nonstandard, naturally) that will ease the life of programmers and sell lots more compilers. The temptation to include these features is certainly not all bad, as it does generate an endless stream of new ideas for compilers, but it feeds directly into the other two forces. The result, if uncontrolled by tough project management and programmer self-discipline, is code that is neither maintainable nor portable.

### 1.2.4 CodeCheck can help!

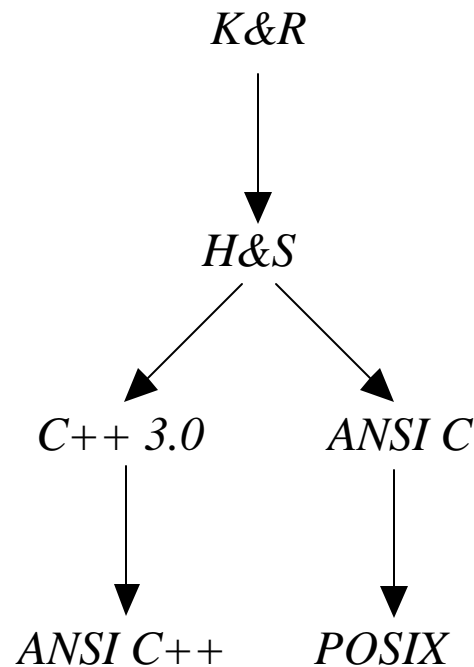
For the C language programmer, to defend against all programming practices that can threaten portability or maintainability is a task requiring both an encyclopædic knowledge of C compilers and an almost superhuman level of self-discipline and attention to detail. For the project leader, the task of enforcing uniform standards for code structure and style is a severe test of the ability to read and critique great volumes of dense code. CodeCheck is designed to automate these tasks:

- The C programmer can use CodeCheck to review his or her code at the end of the day, and to identify questionable constructions that might have crept in. This daily CodeCheck program can implement many *lint*-like code checking operations, as well as checking for adherence to project style specifications.
- The project leader can use a different CodeCheck program on a weekly basis to verify the programmers' adherence to the project style specifications, to quantify the amount of code produced, and to measure critical qualities of the code, *e.g.* density and complexity.
- Software contractors are frequently required to certify that their code conforms to published governmental and industrial standards for code complexity, among them the McCabe and Halstead measures. A CodeCheck program can be run at the conclusion of a project to document these particular measures, and many others too.

## 1.3 Why Programs Fail to Port

### 1.3.1 The Many Standards for the C Language

There seem to be no fewer than four “standards” for the C language, all of which are covered by CodeCheck. Figure 2 depicts the family tree for C standards, with the earliest version on top:



**Figure 2: The Evolution of C Standards.**

Each descendent of the original C has added significant extensions to the original language, while trying to remain true to the spirit of C.

- ? The **K&R** standard, as described in the first edition of Kernighan & Ritchie (1978). This is certainly the single most influential book in the history of C. The language was only loosely defined in this “standard,” however, and it lacks many of the popular features that are commonplace now (e.g. enumerated constants, prototypes, the void type). Although obsolete, there are still many K&R compilers in daily use around the world.

- ? The **H&S** standard, as described in the first edition of Harbison & Steele (1984). This was the first careful description of the K&R standard, with many modern extensions included (*e.g.* the enum and void types). The H&S standard represents a transitional phase between K&R and ANSI. Most pre-ANSI compilers in use today are best described as adhering to the H&S standard.
- ? The **ANSI C** standard, as defined by the American National Standards Institute and certified internationally as ISO/IEC 9899. This version represented a significant advance in precision over H&S. It also introduced several significant innovations (*e.g.* the preprocessor paste operator).
- ? The **POSIX** standard, as defined by the American National Standards Institute and certified internationally as ISO/IEC 9945. Part 1 of this standard includes and extends the ANSI C standard, and details the interface and behavior of a standard library of operating system services.
- ? The **C++ 2.0** standard, as defined in “The Annotated C++ Reference Manual,” by Ellis and Stroustrup (1990). This book is the base document for an ANSI committee that is now developing an official standard for C++.
- ? The **C++ 3.0** standard, as defined in “The C++ Programming Language Manual, **3<sup>rd</sup> Edition**” by Bjarne Stroustrup (1997). This book is the base document of the pending ANSI C++ standard.

### 1.3.2 Two kinds of incompatibility

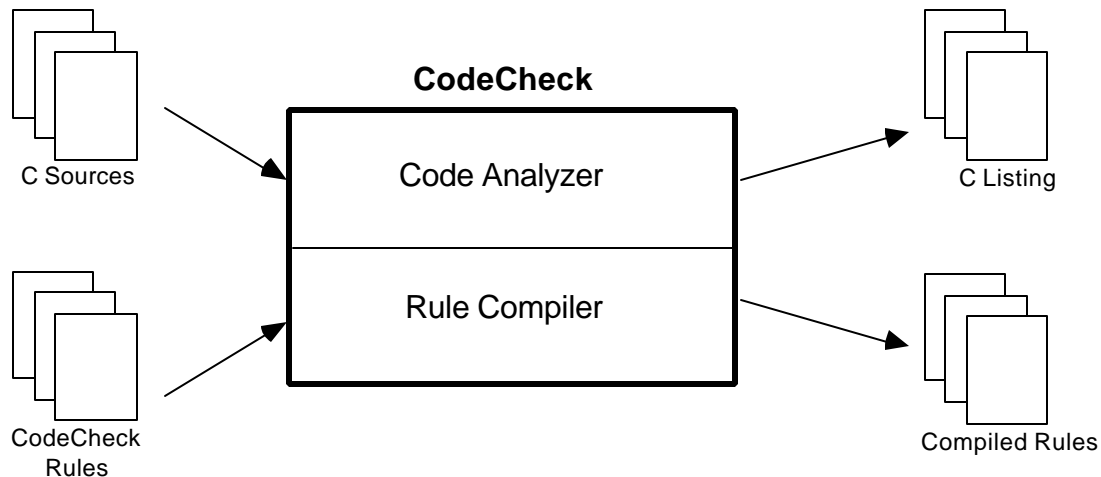
It is useful to break down a portation problem into two separate sources of incompatibility:

1. The source environment will invariably have a variety of idiosyncrasies which are common to no other, and which differ from the ostensible standard on which the environment is based. These differences are **source** portation problems.
2. The target environment will differ somewhat from the standard on which it was based. These differences are **target** portation problems.

The prepackaged CodeCheck rule files supplied by Abraxas with CodeCheck include several that address source and target portation problems. The source portation rule files have names that begin with “from”, as in “fromVax.cc”, which detects special keywords and other peculiarities that are found only on Vax C compilers. The target portation rule files have names that begin with “to”, as in “toKR.cc”, which tests for non-K&R syntax and keywords.

## 1.4 The Structure of CodeCheck

A CodeCheck program looks just like a very simple C program. Indeed, CodeCheck programs are written using a small subset of the C grammar, so anyone who can read C can also read CodeCheck. A CodeCheck program is, in fact, just a collection of if-statements (called “rules”) and variable declarations. The CodeCheck interpreter translates this collection of rules into pseudocode, which is used during the analysis of a C source to control the code checking operation.



**Figure 5: Actions of the two components of CodeCheck.**

To analyze a C source file, the user has only to specify the name of the C source file and the name of the CodeCheck program. The CodeCheck program will be compiled (if necessary), and then the C source file is analyzed in accordance with the CodeCheck rules. As depicted in Figure 5, CodeCheck has two logically separate components — the Code Analyzer and the Rule Compiler.

### A brief bibliographic note

For those who are interested in referring to original sources, this manual makes many references to the C literature. These are given in a compressed format, as illustrated below. Details (title, *etc.*) are given in the bibliography.

HS84:182	<i>means</i>	Harbison & Steele, 1984, page 182.
RJ88:52	<i>means</i>	R. Jaeschke, 1988, page 52.
AK89:99	<i>means</i>	A. Koenig, 1989, page 99.

## 1.5 Debugging with CodeCheck

CodeCheck is, in addition to all of its other functions, a sensitive bug detector, capable of identifying subtle bugs that many compilers miss. A program that compiles without error may still fail to pass CodeCheck's rigorous cross-module syntactic and semantic analyses. There are two common reasons for this: (a) your program deviates from strict C in ways that your compiler permits, or (b) your program actually has a fault whose presence has gone unnoticed. The former case is a mild problem — it only implies a lack of portability — but the latter case may be quite serious.

To use CodeCheck as a bug-detector, use the rule file `error.cc` for an efficient “once-over-lightly” check of your project or any one of its source files. Even with no rule file at all, CodeCheck still performs a tremendous variety of unusual syntactic and semantic checks on its input. Given an entire project, for example, CodeCheck will compare external declarations and macro definitions across files and will advise if any discrepancies are found — regardless of how many or how few rules are included in the rule file. Please note that CodeCheck does *not* perform many of the standard semantic checks that all C and C++ compilers do. CodeCheck is designed to provide error checking that *complements* the checking performed by your compiler.

## 1.6 Predefined Macro Constants

CodeCheck has a predefined macro constant, `CODECHECK`, which is designed to permit conditional checking of C code. This macro constant has the value  $100 \times (\text{CodeCheck version number})$ . Thus in CodeCheck version 8.02 this constant will have the value 802.

The `CODECHECK` macro can be used to hide code from CodeCheck, so that it will not be checked. This is extremely useful, for example, when in-line assembler code is intermixed with C code. Here is an example:

```
#ifndef CODECHECK
...
... /* Code to be hidden from CodeCheck */
...
#endif
```

The `lint` macro can be used in exactly the same way. CodeCheck predefines this macro with the value 2.

CodeCheck also has another predefined macro constant `BETA`, for a version in format `x.xxBy`, `BETA` has value of `y`. For a version in format `x.xx`, the value of `BETA` is 0. Combined with macro `CODECHECK`, you can distinguish different minor releases.

Depending on the specific environment for which it is implemented, CodeCheck will predefine certain additional macro constants. These constants (as of CodeCheck version 8.02 ) are listed in the following table. See the update notes that accompany CodeCheck for the latest additions and changes. The table of macros is organized by operating system and compiler. Any of these may be changed by the user on the command line (using the `-D` or `-U` options) or from within rules (using the CodeCheck *define* and *undefine* functions).

Constant Value Comment

CODECHECK	802	Major Rev 802
BETA	3	Minor Rev 3
lint	2	
__STDC__	1	Option -k2 <b>only</b> .
__STDC__	0	<b>Except</b> option -k2.
__cplusplus	1	C++ <b>only</b> (-k4 through -k9).
cplusplus	1	C++ <b>only</b> (-k4 through -k9).
__FILE__	<file name>	
__LINE__	<line number>	
__DATE__	<date>	
__TIME__	<time>	

Unix Operating System

unix	1	
__unix	1	

Constant Value Comment

DOS Operating System

MSDOS	1	
M_I386	1	
M_I86	1	
M_I86LM	1	
__386__	1	
__I386__	1	
__MSDOS__	1	
__LARGE__	1	<b>Except</b> option -k7 or -k11
__BORLANDC	0x0500	<b>Except</b> option -k7 or -k11
__TURBOC__	0x0500	<b>Except</b> option -k7 or -k11
__WIN32	1	

OS/2 Operating System

__OS2__	1	
__FLAT__	1	
__IBMC__	200	<b>Except</b> option -k6
__IBMCPP__	200	Option -k4 <b>only</b>
__32BIT__	1	<b>Except</b> option -k6
__M_I386	1	<b>Except</b> option -k6

NT Operating System

i386	1	
MSDOS	1	
__M_I386	300	
__MSDOS	1	
__X86__	1	
__WIN32	1	

VMS Operating System

vax	1
vms	1
vaxc	1
vax11c	1
VAX	1
VMS	1
VAXC	1
CC\$gfloat	1
CC\$parallel	1

Macintosh Operating System

applec	1
MC68000	1
mc68000	1
m68k	1
macintosh	1

Borland C++

__BCPLUSPLUS__	0x0340
__TCPLUSPLUS__	0x0340
__CDECL__	1
_Windows	1

<u>Constant</u>	<u>Value</u>	<u>Comment</u>
-----------------	--------------	----------------

---

Borland C++ continued:

__TEMPLATES__	1
wchar_t	short

Microsoft C++

__single_inheritance		Expands to nothing.
__multiple_inheritance		Expands to nothing.
__virtual_inheritance		Expands to nothing.
_M_I86	1	<b>Except</b> Windows NT.
_M_I86LM	1	<b>Except</b> Windows NT.
_M_IX86	300	
_MSC_VER	1200	
_MSDOS	1	
_X86_	300	
i386	1	
MSDOS	1	
_WIN32	1	

Metaware High C

__HIGHC__	1
-----------	---

Symantec C++

__SC__	700
--------	-----

IBM Visual Age C++  
\_\_IBMCPP\_\_ 350

Metrowerks CodeWarrior  
\_\_MWERKS\_\_ 1

Debugging your source code with the CodeCheck preprocessor can be greatly enhanced by using the "-D?" switch, which will display the current state of CodeCheck internal symbol table for the pre-processor. If a particular intrinsic definition is non-desirable then the "-U" switch can be used to undefine the macro.

The CodeCheck product does not include the C/C++ system header files ( `stdio.h`, `iostream.hpp`, ... ). These must be obtained from your compiler vendor, e.g. if source code to be analyzed by CodeCheck explicitly references `stdio.h` ( `#include <stdio.h>` ), then that header file must be available for CodeCheck to analyze. In summary for CodeCheck to analyze source code, all parts of the entire project to be analyzed must be present in order for the analysis to be successful.

# Chapter 3: Checking Types

This section describes how to use CodeCheck to analyze type information with CodeCheck rules. These techniques are necessary for those who wish to write their own CodeCheck rules which detect conditions that depend on type information.

The ability to detect and analyze type information in declarations has been a part of CodeCheck since version 4.03. However, the ability to detect and analyze type information within *executable* code is relatively new to CodeCheck, having been introduced for C in version 5.04, and for C++ in version 5.05. Please refer to the CodeCheck Reference Manual for the exact definitions of the CodeCheck variable and functions mentioned in this chapter, and the Abraxas Technical Note ( [www.abraxas-software.com/TechNotes.html](http://www.abraxas-software.com/TechNotes.html) ) series for recent changes and enhancements.

There are five broad categories of type-related rules that CodeCheck can enforce. Rules can be written that detect:

1. a declaration of any specified type,
2. a cast to or from any specified type,
3. an implicit type conversion to or from any specified type,
4. use of a variable or function of any specified type,
5. use of an operand of any specified type for any specified operator.

In addition, CodeCheck automatically checks function argument types for compatibility with the prototype for the function called, if one is in scope, and also checks the function return value for compatibility with the declared function return type.

## 3.1 How to Analyze a Type Declaration

Types in C and C++ are either simple or complex. A simple type consists of an unmodified base type, *e.g.* `int` or `float`, with possible qualifiers such as `const`, `volatile`, `near`, `far`, `huge`, `export`, *etc.* A complex type has a

simple type as its base, and in addition has one or more additional levels, *e.g. pointer to..., array of..., function returning..., or reference to...* (the latter is allowed only in C++ and a few nonstandard C dialects). Each of these levels may have also have qualifiers (*e.g. const, pascal, interrupt, etc.*).

When an identifier is declared, CodeCheck set the variable `dcl_levels` to the number of levels in the type. Thus for simple variables `dcl_levels` will be zero. CodeCheck sets the variable `dcl_base` to an integer that identifies the base type of the identifier. The possible values of `dcl_base` are defined as manifest constants in the CodeCheck header file `check.cch`, which should be included in every rule file that makes use of type-checking services. Here are the first five base types from `check.cch`:

```
#define VOID_TYPE      1
#define CHAR_TYPE     2
#define SHORT_TYPE    3
#define INT_TYPE      4
#define LONG_TYPE     5
```

As an example of the use of these CodeCheck variables to detect a specified type, here is a rule that will issue a message whenever a simple variable of type `char` is declared:

```
if ( dcl_base == CHAR_TYPE )
    if ( dcl_levels == 0 )
        warn( 1234, "Variable %s is a char.", dcl_name() );
```

When the type in a declaration is complex, the function `dcl_level()` returns an integer that identifies the kind of each level. Here is an example rule that prints out the type of every global or local identifier that is declared:

```
int i, kind;

if ( dcl_global || dcl_local )
{
    printf( "Variable %s:  ", dcl_name() );
    i = 0;
    while ( i < dcl_levels )
    {
        kind = dcl_level( i++ );
        switch( kind )
        {
            case ARRAY:
                printf( "array of " );
                break;
            case POINTER:
                printf( "pointer to " );
                break;
        }
    }
}
```

```

    case REFERENCE:
        printf( "reference to " );
        break;
    case FUNCTION:
        printf( "function() returning " );
        break;
    }
}
printf( "%s\n", dcl_base_name() );
}

```

The manifest constants `ARRAY`, `POINTER`, `REFERENCE`, and `FUNCTION` are defined in `check.cch`. In addition to `dcl_level()`, this rule also used the functions `dcl_name()` and `dcl_base_name()`. These functions return the declarator name and the name of the base type of the declaration, respectively. The variables in the trigger for this rule are `dcl_global` and `dcl_local`, which CodeCheck sets to 1 when a global or local identifier is declared, respectively. (In this context “global” means file scope and external linkage, while “local” means function or block scope.)

To obtain the type qualifiers for each level of a type, including the base type level, use the function `dcl_level_flags()`. This function takes as its argument the level, just like `dcl_level()`, and returns an integer that has a bit set for each qualifier present. The header file `check.cch` contains manifest constants that can be used as masks to obtain each of these qualifier flags. For example, the following rule can be used to detect declarations in which the first level has the `const` qualifier:

```

if ( dcl_level_flags(0) & CONST_FLAG )
    warn( 1234, "%s has been declared constant.", dcl_name() );

```

There are many other declarator variables and functions that are useful for analyzing declared types — refer to the CodeCheck Reference Manual for details.

## 3.2 How to Determine the Type of an Identifier

The CodeCheck variables and functions for determining the type of an identifier that is used within executable code are almost identical to those for declarations, except that they carry the prefix `idn_` instead of `dcl_`.

The function `idn_filename()` and the variable `idn_line` can be used to determine the location (*i.e.* file name and line number) of the declaration that is currently in scope for the identifier as it is used in the executable code.

The variables `idn_global`, `idn_local`, `idn_member`, and `idn_parameter` also resemble their `dcl_` counterparts: they are used to determine whether the identifier has global, local, or class scope, or is a function parameter, respectively.

### 3.3 How to Determine the Type of an Operand

The CodeCheck variables and functions for determining the types of all the operands of executable operators are similar to their counterparts for declarations and identifiers. The major difference is that they require an additional argument which specifies which operand to describe.

Operators may be unary (one operand, *e.g.* `~`), binary (two operands, *e.g.* `+=`), or ternary (three operands, *e.g.* `?:`). Functions have as many operands as they have arguments. Whenever an executable operator is encountered, CodeCheck sets the appropriate `op_` variables to indicate which operator was found, and sets `op_operands` to the number of operands taken by this operator.

The CodeCheck functions `op_base()`, `op_levels()`, `op_level()`, and `op_level_flags()` differ from their declarator and identifier counterparts in only one way: their first argument specifies which operand is to be described. **The operands are indexed from right to left.** Thus the rightmost operand is the operand 1, while the leftmost operand is the last (index given by `op_operands`). For example, in the expression `x = a + b` the first operand for `op_add` is `b`, and the second is `a`. For the operator `op_assign`, the first operand is the result of `(a+b)`, and the second is `x`.

In the special case of the cast operator, the first operand is the type of the value to be cast, while the second operand is the result type after the cast has been performed. Here is an example rule that detects every cast of a pointer to a struct XYZ, to any result type, *i.e.* a cast that looks like `(struct XYZ *)`:

```
if ( op_cast )
{
    if ( (op_levels(1) == 2)                &&
        (op_level(1,0) == POINTER)        &&
        (op_base(1) == STRUCT_TYPE)       &&
```

```

        (strcmp(op_base_name(1),"XYZ") == 0) )
warn( 1234, "Cast from (struct XYZ *) to anything." );
}

```

## 3.4 How to Detect Implicit Type Conversions

Implicit type conversions can happen in three different contexts. First, when a value of one type is assigned to a value of another type, without an explicit cast, then an implicit type conversion takes place. Second, when the type of an argument that is passed to a function differs from the type of the formal parameter in the prototype for the function, then it must be implicitly converted. Third, an implicit type conversion occurs when type of the value in a return statement differs from the return type of the function.

The CodeCheck functions used for determining the types involved in all implicit type conversions are *the same functions* as used for operators. However, to detect an implicit type conversion, one of the `cnv_` variables must be used as the trigger in the rule. When one of these variables is set, then CodeCheck uses the `op_` functions *as though a cast operator were present* for the conversion.

Here is an example rule that detects every implicit conversion of anything to a pointer to a struct XYZ:

```

if ( cnv_any_to_ptr || cnv_ptr_to_ptr )
{
  if ( (op_levels(2) == 2)                &&
        (op_level(2,0) == POINTER)       &&
        (op_base(2) == STRUCT_TYPE)      &&
        (strcmp(op_base_name(2),"XYZ") == 0) )
    warn( 1234, "Implicit conversion to (struct XYZ *)." );
}

```

# Chapter 4: Portable Style

This section describes how to use CodeCheck to monitor style for portability. Many of the rules described here are based on internal corporate standards for C coding that have been made available to Abraxas Software, and also on the recommendations of two influential books: **C Programming Guidelines, Second Edition**, by Thomas Plum, and **Portability and the C Language**, by Rex Jaeschke.

The guidelines and recommendations found in these sources were designed primarily to enforce both portability and maintainability. In the subsections that follow, those guidelines that primarily affect program portability are presented. Each guideline is followed by a description of the relevant CodeCheck variables that can be used to construct corresponding CodeCheck rules.

## 4.1 Lexical Issues in Portability

### 4.1.1 Lexical Rules for Variable Names

C programmers have evolved a great variety of lexical guidelines for variables and their declarations. The guidelines reported here are specifically intended to ensure portability. Additional guidelines that are intended to improve maintainability may be found in Chapter 4.

1. Spell variable names in lower case only.
2. Names with external linkage must be unique in their first 6 characters.
3. Do not begin an identifier with an underscore character.

The predefined CodeCheck variables which are used to detect violations of the above guidelines are, respectively:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
------------------------	-----------------------

<code>dcl_any_upper</code>	Set to 1 if an upper-case character is found in an identifier name when it is declared.
<code>dcl_extern_ambig</code>	If two external identifiers have names that agree on the first 6 or more characters, <i>regardless of case</i> , then this variable is set to the number of consecutive characters on which they agree.
<code>dcl_underscore</code>	Set to 1 if the name of a declared identifier begins with an underscore character.

### 4.1.2 Nonstandard Characters

Many C compilers allow the use of characters that are not in the standard C character set. Nonstandard characters are, by definition, not portable. The standard characters are (HS88:6):

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
! # % ^ & * ( ) - _ + = ~ [ ] \ | ; : ' " { } , . < > / ?
blank      newline      backspace      horizontal-tab
vertical-tab  form-feed      carriage-return
```

*Recommendation:* For general portability, do not use nonstandard characters. Note that the characters \$ and @ are nonstandard. CodeCheck provides a predefined variable with which to detect nonstandard characters:

<b>Variable</b>	<b>Meaning</b>
<code>lex_nonstandard</code>	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.

To ignore nonstandard identifiers, call function `skip_nonansi_ident()`.

**Function****Meaning**

`skip_nonansi_ident( char )` Skip non-ANSI identifiers beginning with '@','\$' or '\'. 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.

**4.1.3 Trigraphs**

Trigraphs are special 3-character sequences introduced in ANSI C. Trigraphs are significant as a portability issue only to the extent that older programs which unwittingly use trigraph sequences within literal strings will no longer compile correctly (RJ:28). ANSI C compilers translate trigraph sequences into their corresponding single ASCII characters at a very early stage in the lexical analysis phase of compilation. The trigraph sequences and their corresponding ASCII characters are:

<u>trigraph</u>	<u>meaning</u>
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	
??>	}
??-	~

*Recommendation:* Search old programs for the ?? symbol pair, and replace every occurrence with ?\?. Older compilers will simply ignore the backslash, while ANSI compilers will treat ?\? as a question mark followed by an escape sequence, thus recoding the string to ??. CodeCheck provides two predefined variables for identifying inadvertent trigraphs:

**Variable****Meaning**`lex_trigraph`

Set to 1 if an ANSI trigraph is found.

`lex_str_trigraph`

Set to 1 if a trigraph is found within a string literal.

#### 4.1.4 Numeric Escape Codes

Numeric escape codes are permitted in C, but they are normally used to refer to control characters in the ASCII character set (HS84:25). Such numeric escape codes will cause a program to fail when it is compiled and used in a non-ASCII (e.g. EBCDIC) environment.

There is a potentially confusing aspect of hexadecimal escape sequences. Unlike octal sequences, which may have at most three digits, the ANSI C standard specifies that a hexadecimal escape sequence may have any number of digits. Thus the string literal `"/xabcd"` contains only one character (because a, b, c, and d are all valid hex digits).

*Recommendation:* Do not use numeric escape codes unless it is absolutely necessary. Carefully document the meaning of each such usage in the source code, and use a manifest constant (`#define`) to make its meaning apparent. CodeCheck provides two predefined variables for detecting numeric escape codes:

<b>Variable</b>	<b>Meaning</b>
<code>lex_hex_escape</code>	When a hexadecimal escape sequence is found, this variable is set to number of hexadecimal digits found.
<code>lex_num_escape</code>	When a non-zero numeric escape sequence is found, the value of this variable is set to the value of the escape sequence.
<code>lex_zero_escape</code>	When a zero escape sequence is found (e.g. <code>\0</code> , <code>\00</code> , <code>\x0</code> , <code>\0x0</code> ), the value of this variable is set to 1 if the context is a character literal, or 2 if the context is a string literal.

#### 4.1.7 Escape Sequences in Character and String Literals

An escape sequence within a character or string literal is signaled by the backslash character: `\`. The unrestricted use of escape sequences is a frequent source of portability problems.

1. For reasons that shall remain forever mysterious, many pre-ANSI compilers allow the digits 8 and 9 to appear within an octal escape sequence, as in

\09. This usage has always been hopelessly confusing, has always been non-portable, and is now, fortunately, also ungrammatical.

2. Many pre-ANSI compilers do not support the hexadecimal escape sequence, as in \xA3. This usage is therefore not portable except among ANSI compilers. The hexadecimal escape sequence \0xA3 (with a zero appearing before the x) is even rarer, and is forbidden by ANSI.

3. In the K&R dialect of C the only defined escape characters are in this set:

```
\n  \b  \t  \r  \f  \\  \"  \'
```

In all other cases when a backslash is followed by a character, the backslash is ignored. The H&S dialect allows one more escape, \v (vertical tab). The ANSI standard includes three further escape characters: \a (alert or bell), \x (hexadecimal), and \? (to disambiguate trigraphs). Thus pre-ANSI programs which rely on the backslash being ignored before the letters a, v, or x are no longer portable.

4. The empty character constant formed by two successive single quote marks (' ') is not consistently interpreted by C compilers, and is not allowed by many. In particular, it is not always the same as the null character '\0'.

5. Many C compilers allow character constants to have more than one character. Even if all compilers did allow this usage, it would still be non-portable due to the infamous NUXI problem. ("NUXI" refers to the worst-case scrambling of the string "UNIX" that can result when porting encoding character strings).

To detect the above portability problems, CodeCheck provides the following six predefined variables:

<b>Variable</b>	<b>Meaning</b>
<i>lex_big_octal</i>	Set to 8 or 9, respectively, when a numeric escape sequence or octal integer contains the digits 8 or 9.
<i>lex_hex_escape</i>	When a hexadecimal escape sequence is found, this variable is set to number of hexadecimal digits found.
<i>lex_ansi_escape</i>	Set to 1 if an escape sequence contains one of the new ANSI escape characters: a, v, or ?.
<i>lex_not_KR_escape</i>	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.
<i>lex_char_empty</i>	Set to 1 if an empty character constant is found (e.g. ""). This variable does <i>not</i> flag the null character constant ('\0').
<i>lex_char_long</i>	Set to 1 if a character constant is longer than one character.

### 4.1.8 System Variables

A useful convention has evolved within the C community in which identifiers that are defined and used by the system (*i.e.* variables used by the compiler, the linker, or standard system header files) are spelled with a leading underscore character. If programmers conform to this convention by never spelling an identifier in this way, then name conflicts are prevented. Unfortunately, some programmers are unaware of this convention, and may inadvertently spell an identifier with a leading underscore. Even if the program compiles without error, it will break as soon as it is compiled on another system that happens to use one of these names. This is, therefore, an obscure but significant portability problem.

*Recommendation:* Adhere to this convention religiously. Do not spell identifiers with a leading underscore character, unless you are writing system code. CodeCheck predefined variable:

<b>Variable</b>	<b>Meaning</b>
<i>dcl_underscore</i>	Set to 1 if an identifier begins with an underscore character.

### 4.1.9 The Numeric Constant Suffixes U, F, and L

The ANSI standard and some pre-ANSI compilers allow a suffix of U (for unsigned) or F (for float) on numeric constants, in precisely the same way that the

suffix L (for long) is allowed in K&R C. Needless to say, this is not a portable usage among pre-ANSI compilers.

The suffix L is generally portable, except when it is used on a floating constant. Only some non-ANSI compilers recognize the long float type, so this is a non-portable use of the L suffix.

*Recommendation:* The new suffixes U and F solve many problems of numeric ambiguity, and *should be used* on the grounds that they increase program clarity and make maintenance easier. The cost in terms of lack of portability is substantially less than the benefit for program clarity.

CodeCheck provides four predefined variables for suffix detection:

<b>Variable</b>	<b>Meaning</b>
<code>lex_float</code>	Set to 1 if a numeric constant is found with the suffix 'F' or 'f'.
<code>lex_long_float</code>	Set to 1 if a floating constant is found with the suffix 'L' or 'l'.
<code>lex_suffix</code>	Set to 1 if a numeric constant is found with any suffix ('F', 'f', 'L', 'l', 'U', or 'u').
<code>lex_unsigned</code>	Set to 1 if a numeric constant is found with the suffix 'U' or 'u'.

#### **4.1.10 Octal and Hexadecimal Numeric Constants**

Octal numeric constants have always been among the very worst features of the C language, and the ANSI standard does little to improve matters. That octal constants should be distinguished from decimal constants simply by adding a leading zero is just plain appalling. Uncounted thousands of hours of programmer time have been invested over the years in finding errors caused by plausible but wrong numeric constants. This is a serious and continuing maintenance problem.

Hexadecimal numeric constants are in themselves problematic, but difficulties arise because they are frequently used to refer to characters. This use of nu-

meric constants is non portable because C is not tied to the ASCII character set. As one important example, bear in mind that IBM mainframes use the EBCDIC character set.

*Recommendation:* Avoid octal constants whenever feasible. If not feasible, add a comment that loudly proclaims that the constant is intended to be octal, or use a manifest constant (e.g. `#define OCTALTEN 010`) to make plain the octal nature of the number. Similarly, avoid the use of any numeric constant to refer to a character. CodeCheck provides a predefined variable for detecting constants of any radix:

<b>Variable</b>	<b>Meaning</b>
<code>lex_radix</code>	Set to the radix of an integer constant (2, 8, 10, or 16).

#### **4.1.11 The End-of-File Marker**

The ANSI standard is quite clear on its specification of the appearance of the end of a file: the last character before the end-of-file marker in a non empty source file must be a newline character. This makes good sense, but some pre-ANSI compilers allow a file to terminate with any character. Consequently, the ends of files must be checked for portability to ANSI. This may seem like a minor point, but some programmers have used this deficiency of pre-ANSI compilers to accomplish some very bizarre and completely unnecessary tricks.

*Recommendation:* It is good practice to end a file with a newline, and it is bad practice to play games with header files that end in the middle of macro definitions, declarations, or function definitions.

<b>Variable</b>	<b>Meaning</b>
<code>lex_nl_eof</code>	Set to 1 if a non empty source file does not end with a newline.

### 4.1.12 String Concatenation

Under the ANSI C standard, consecutive string constants are concatenated automatically during the lexical analysis phase of compilation. Older compilers may not perform this operation, thus causing a problem for code that is to be ported from ANSI C to an older compiler. CodeCheck predefined variable:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>lex_str_concat</i>	Set to 1 if two string constants are found, separated only by whitespace.

### 4.1.13 Embedded Assembler Code

Assembler code is, almost by definition, non portable. CodeCheck provides a predefined variable with which large source files can be automatically scanned for embedded assembler.

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>lex_assembler</i>	Set to 1 when assembler code is detected.

### 4.1.14 Continuation Lines

In ANSI standard C, a line can be continued with a backslash character that is followed immediately by a newline character. In older dialects of C this kind of line continuation marker was permitted only within macro definitions. CodeCheck provides a predefined variable for detecting the use of the continuation marker outside of macro definitions.

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>lex_backslash</i>	Set to 1 if a backslash-newline pair is found at the end of a line that is not part of a macro definition.

#### 4.1.15 Special Reserved Keywords and Identifiers

Many compilers have special keywords and identifiers which are shared by no other compiler. For example, Microsoft uses the special keyword `_based`. Needless to say, these keywords are generally non portable. CodeCheck provides two functions for detecting specified identifier and keyword names.

<b>Function</b>	<b>Meaning</b>
<code>identifier(char *)</code>	Returns the integer value 1 whenever an identifier is found that matches the argument string.
<code>keyword(char *)</code>	Returns the integer value 1 whenever a keyword is found that matches the argument string.

#### 4.1.16 Wide String and Character Literals

The ANSI standard provides a mechanism for using strings and characters that come from very large alphabets, such as Chinese. These “wide” strings and characters are lexically signaled with the prefix `L` immediately before the leading quote.

CodeCheck provides a predefined variable for detecting wide literals:

<b>Variable</b>	<b>Meaning</b>
<code>lex_wide</code>	Set to 1 if an ANSI wide string or character constant is found (prefix <code>L</code> ).

#### 4.1.17 Visibility of Nested Tag Names

ANSI C and all versions of C++ allow tag definitions to be nested within tag definitions. However, the visibility of identifiers associated with this tag depends on which dialect is in use. In ANSI C the nested tag is treated as though it has global scope. In C++ the nested tag has class scope, *but* in versions prior

to 3.0 the tag name and all of its enum constants are exported to the enclosing scope, to provide compatibility with C. In version 3.0 this compatibility feature is removed for nested enum constants and applies to nested tag and typedef names only if they do not conflict with names in the enclosing scope. Needless to say, a portability problem arises.

*Recommendation:* When writing C++ or porting C code to C++, all nested identifiers should be explicitly scoped when used outside the enclosing scope. Do not depend on compatibility with ANSI C.

CodeCheck provides a predefined variable for detecting problematic unscoped identifiers:

<b>Variable</b>	<b>Meaning</b>
<code>lex_invisible</code>	Set to 1 when an <i>unscoped</i> identifier is visible to ANSI C and all versions of C++ prior to 3.0, but is invisible to C++ 3.0.

## 4.2 Preprocessor Considerations

The preprocessor may very well be the most fertile source of portability problems in the C language. This has occurred because (a) until the ANSI standard, the preprocessor was never adequately standardized, and (b) it is a powerful tool with many quirks and foibles, just crying out for exploitation by “clever” programmers.

### 4.2.1 Whitespace within Preprocessor Directives

Some C preprocessors allow whitespace (space or tab characters) to precede the # symbol, and to come between the # symbol and the preprocessor command. Such use of whitespace is not portable to older compilers, although the trend is toward permitting it (HS88:27). The ANSI standard permits this use of whitespace.

Other forms of whitespace, *e.g.* vertical tabs, form-feeds, backspace characters, *etc.*, are not at all portable if used anywhere within a preprocessor directive.

*Recommendation:* To enhance visibility, preprocessor lines should be clearly marked with the # symbol in the first position of the line. However, indentation of the preprocessor directive after the # symbol is desirable to indicate, for example, the nesting level of *#if* directives. CodeCheck provides three pre-defined variables for detecting whitespace within preprocessor directives:

<b>Variable</b>	<b>Meaning</b>
<i>pp_white_before</i>	Set to the amount of whitespace (in characters) that precedes the # character in a preprocessor directive.
<i>pp_white_after</i>	Set to the amount of whitespace (in characters) that is found after the # character and before the keyword in a preprocessor directive.

*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.

## 4.2.2 Leading Whitespace within Included File Names

Some C preprocessors do not automatically delete any whitespace characters (spaces or tabs) that precede the filename within *#include* directives. For example:

```
#include < wrong.h>
#include " wrong.h"
```

This is a portability issue, since many preprocessors do not perform this service.

*Recommendation:* Avoid leading whitespace in these filenames.

CodeCheck provides a predefined variable for detecting leading whitespace within *#include* filenames:

<b>Variable</b>	<b>Meaning</b>
<i>pp_include_white</i>	Set to 1 if the filename in an <i>#include</i> directive has leading whitespace.

## 4.2.3 Preprocessor Arithmetic

Some preprocessors will not perform any arithmetic at all within preprocessor directives, although they will do logical comparisons. Preprocessor arithmetic is therefore not portable. Even preprocessors that do perform arithmetic (notably ones that conform to the ANSI standard) will still not evaluate the *sizeof* function. This means that some apparently good code ideas, such as the following from a Symantec header, will fail:

```
1  #if sizeof(int) < 4
2      long    total;
3  #else
4      int     total;
```

5 #endif

Recommendations: (a) Avoid preprocessor arithmetic. (b) You may be able to use a definition in the file `limits.h` (supplied with all ANSI compilers) to deduce “sizeof” information. For example, if `INT_MAX` is defined in this file as 32767, then the size of an `int` must be 2 bytes.

CodeCheck provides two predefined variables for detecting these problems:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<code>pp_arith</code>	Set to 1 if a preprocessor directive requires arithmetic calculation.
<code>pp_sizeof</code>	Set to 1 if a preprocessor directive requires the evaluation of a <code>sizeof</code> function.

#### 4.2.4 Macro Parameters

CodeCheck automatically checks for certain common problems that can occur with macro parameters. For example, a macro call with the wrong number of arguments will evoke an automatic CodeCheck warning message. Whenever this occurs CodeCheck will also set the predefined variable `lex_bad_call` so that the user can have his own custom-designed error message triggered. A few deviant preprocessors allow a macro call to have fewer arguments than specified, even though this is permitted in neither K&R nor ANSI C.

A separate problem occurs when a variable has the same name as a macro function. Most compilers will accept this usage, but some will treat it as an error.

CodeCheck provides four predefined variables for detecting these and other similar problems:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<code>lex_bad_call</code>	Set to the difference between the number of arguments found and the number of arguments expected when a macro function is expanded.
<code>lex_null_arg</code>	Set to 1 if an actual argument is omitted in a macro call, e.g. <code>XYZ(abc, , 123)</code> .

<code>pp_arg_count</code>	Set to the number of formal parameters found in a macro definition.
<code>pp_empty_arglist</code>	Set to 1 if the definition of a macro “function” has no formal parameters.
<code>pp_overload</code>	Set to 1 if a variable name conflicts with a macro function name.

#### 4.2.5 Comments in Macro Definitions

Once upon a time there was a marvelously clever programmer who realized that by placing a comment in a macro definition like this

```
#define PASTE(a,b)  a/**/b
```

he could cause the preprocessor to “paste” the actual arguments for `a` and `b` together, so that the compiler would see only the single token `ab`. For example, the macro call `f(PASTE(i,j))` would be perceived by the compiler as `f(ij)`, regardless of the values of the variables `i` and `j`. From this little piece of cleverness there sprang up a whole cottage industry of token pasting by programmers eager to exploit the very latest in program obfuscation. This example, which is dissected further in RJ88:37, violates the cardinal principle of program maintainability: ***the intent of the programmer must be fully evident on first reading***. Furthermore, not all compilers will be so accommodating as to ignore the embedded comment—all ANSI preprocessors will treat the comment as whitespace, and will therefore not paste any tokens together. Thus this mode of token pasting also presents a portability problem.

Fortunately, ANSI has come to the rescue with two badly needed provisions: (a) comments within macro definitions must be treated as whitespace, and (b) programmers who feel an overwhelming need to paste tokens may use the new “paste” operator (`##`) for the preprocessor.

*Recommendation:* Do not paste tokens unless you have a truly solid reason. CodeCheck provides two predefined variables for detecting these problems:

<b>Variable</b>	<b>Meaning</b>
<code>pp_comment</code>	Set to 1 if two tokens within a macro definition are separated only by a comment.
<code>pp_paste</code>	Set to 1 if the “paste” operator (##) is found in a macro definition.

In its normal (ANSI-conforming) mode of operation, CodeCheck treats any comment within a macro definition as whitespace. However, when the **-K0** command-line switch is active (indicating K&R [1978] C syntax), CodeCheck interprets an embedded comment as a paste operation. Thus even old programs that use this unfortunate feature can be scanned successfully with CodeCheck.

#### 4.2.6 The `#elif` Directive

The ANSI standard and some pre-ANSI compilers allow the `#elif` preprocessor directive and the `defined()` preprocessor function. These useful additions to the preprocessor permit clearer conditional compilation directives, as this example illustrates:

```

1  #if defined( macintosh )
2      .
3      .
4  #elif defined( MSDOS )
5      .
6      .
7  #elif defined( vms )
8      .
9      .
10 #endif

```

Unfortunately, not all compilers support these new constructions.

*Recommendation:* If maintainability is more important than portability, then by all means use these new constructions — the benefits are significant. Avoid these new constructions only if you are writing for the most general kind of portability.

CodeCheck provides two predefined variables for detecting these usages:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>pp_defined</i>	Set to 1 if the <code>defined</code> preprocessor function is encountered.
<i>pp_elif</i>	Set to 1 if the <code>#elif</code> preprocessor directive is encountered.

#### **4.2.7 Macro Names and Parameters Inside Strings**

Some older compilers permit macro expansion within string literals. This practice is permitted neither in ANSI C nor in the majority of non-ANSI compilers. It can pose a serious portability problem for programs that rely on such expansions.

Another problem occurs in macro definitions when a formal parameter of a macro is used inside a string literal. The Vax C compiler, among others, will perform substitution for the formal parameter name inside the string literal, but ANSI standard compilers will not.

CodeCheck provides two predefined variables for detecting these problems:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>lex_str_macro</i>	Set to 1 when a macro identifier is found within a string constant.
<i>pp_arg_string</i>	Set to 1 if a macro formal parameter is found inside a string literal in the macro definition.

#### **4.2.8 New ANSI Preprocessor Features**

In the process of standardizing and regularizing the C preprocessor, the ANSI standards committee elected to introduce a number of new features. Each of these features constitutes a portability problem for developers who wish to port

their project from an ANSI conforming compiler to a strict K&R compiler. These new features include the following:

1. Use of a macro that expands to a filename within an `#include` directive.
2. The `#error` and `#pragma` preprocessor directives.
3. The `defined()` preprocessor function.
4. The `paste` and `stringize` preprocessor operators.
5. The `#elif` conditional directive.

In addition to predefined variables which flag each new feature, CodeCheck provides a single predefined variable, `pp_ansi`, for detecting all such features, and `pp_unknown` for flagging non-ANSI preprocessor directives.

<b>Variable</b>	<b>Meaning</b>
<code>pp_ansi</code>	Set to 1 if a preprocessor feature is encountered that is new with the ANSI standard.
<code>pp_defined</code>	Set to 1 if the <code>defined</code> preprocessor function is encountered.
<code>pp_elif</code>	Set to 1 if the <code>#elif</code> preprocessor directive is encountered.
<code>pp_error</code>	Set to 1 if the <code>#error</code> preprocessor directive is encountered.
<code>pp_include</code>	Set to the following on an <code>#include</code> directive: 1: filename is in quotes, and was obtained from a macro expansion, 2: filename is in quotes, <b>not</b> from a macro, 3: filename is in angle brackets, from a macro, 4: filename is in angle brackets, <b>not</b> from a macro. 5: filename is not enclosed (Metaware C <i>only</i> ). 6: filename is not enclosed (Vax C <i>only</i> ).
<code>pp_paste</code>	Set to 1 if the ANSI “paste” operator ( <code>##</code> ) is found in a macro definition.

<i>pp_pragma</i>	Set to 1 if a #pragma preprocessor directive is encountered.
<i>pp_stringize</i>	Set to 1 if the ANSI “stringize” operator (#) is found in a macro definition.
<i>pp_unknown</i>	Set to 1 if a non-ANSI preprocessor directive is found.

### 4.2.9 Preprocessor Keyword Substitution

Some preprocessors allow macro names to be used after the # symbol, as in this example:

```
#define macro      define
#define square(x) ((x)*(x))
```

Needless to say, this is not a portable use of macro substitution. CodeCheck provides a predefined variable for detecting this usage:

<b>Variable</b>	<b>Meaning</b>
<i>pp_sub_keyword</i>	Set to 1 if the keyword in a preprocessor directive is itself a macro name.

### 4.2.10 Recursive Macros

A macro is recursive if its name appears as a token within its definition. C preprocessors vary in their permissiveness with respect to recursive macros. The ANSI standard specifies that the definition of a macro be “turned off” for the duration of the expansion of that macro, to prevent the preprocessor from plunging into an infinite recursion. The Microsoft C preprocessor, however, only turns off the definition of macro constants, thus allowing recursive macro functions. Other preprocessors only turn off the current macro being expanded, so that two mutually recursive macros can cause an infinite recursive death.

*Recommendation:* Avoid recursive macros.

CodeCheck provides a predefined variable for detecting recursive macro definitions:

<b>Variable</b>	<b>Meaning</b>
<i>pp_recursive</i>	Set to 1 if a recursive macro definition is found.

#### **4.2.11 Macro Definition Length**

Some preprocessors store each macro definition as a string, usually after whitespace has been eliminated. Such preprocessors may have a maximum, e.g. 100 characters, for the length of any one macro definition. ANSI C, by contrast, does not impose any length restrictions on macro definitions. Thus long macro definitions may cause a problem when porting to non-ANSI compilers.

<b>Variable</b>	<b>Meaning</b>
<i>pp_length</i>	Set to the length (in characters) of the body of a macro definition, after redundant whitespace has been eliminated.

#### **4.2.12 Relative Pathnames for Header File Inclusion**

An ambiguity arises when an `#include` directive within a header file contains a relative pathname. Here is a Unix example:

```
#include <../../foobar.h>
```

Is the file `foobar.h` to be found in a location that is relative to the source file, or relative to the header file that contains the above `#include` directive? Older K&R compilers always look in a location that is relative to the source file. The ANSI standard does not specify where to look. Many modern compilers look in a location that is relative to the header file that contains the `#include` directive. (CodeCheck looks in both places.) This ambiguity means that C and C++ programmers who use relative pathnames may encounter difficulties when changing to a different compiler, or when porting to a different system.

*Recommendation:* Avoid relative pathnames in header #include directives.

CodeCheck provides a predefined variable for detecting relative pathnames:

<b>Variable</b>	<b>Meaning</b>
<i>pp_relative</i>	Set to 1 when an #include directive within a header file specifies a relative pathname.

## 4.3 Portability in Declarations

### 4.3.1 Array, Structure, and Union Initializers

Some new compilers permit initializers for automatic (*i.e.* not *static* and not external) arrays, structures, and unions. What they are actually doing is generating executable code that assigns initial values to these objects. Note that this is somewhat different from true initialization, in which the initial values are part of the image of the program. In any event, since not all compilers permit this usage, it is not portable.

Another difficulty is caused by union initializers in general: many compilers do not allow any sort of union initialization.

*Recommendation:* For the sake of portability, do not try to give initializers to automatic arrays, structures, or unions. It is better either (a) to make such objects that need initializers *static*, or (b) to use explicit initialization. CodeCheck provides two predefined variables for detecting initialization problems:

<b>Variable</b>	<b>Meaning</b>
<code>dcl_auto_init</code>	Set to 1 if an initializer for an automatic array, struct, or union is found.
<code>dcl_union_init</code>	Set to 1 when a union initializer is found.

### 4.3.2 Enumerated Variables

Traditional K&R compilers do not support enumerated variables, but most modern H&S compilers do. Unfortunately, pre-ANSI C compilers are not fully consistent in their implementations. Because of this there is a mild portability problem with enumerated variables.

A greater difficulty is presented by compilers that allow computed values for enumerated constants: these are seldom portable.

*Recommendation:* The advantages of enumerated variables for enhancing program clarity greatly outweigh the portability problems that they may cause. Use them, but don't use computed explicit values for the enumeration constants.

```
1  if ( dcl_init_arith && (dcl_base == ENUM_TYPE) )
2      warn( 99, "Non portable computed initializer." );
```

<b>Variable</b>	<b>Meaning</b>
<i>dcl_init_arith</i>	Set to 1 when a computed initializer is found, or when a computed explicit value for an enumerated constant is found.
<i>dcl_base</i>	Set to an integer which identifies the base type of the current declarator. The base types are defined as manifest constants in the header file <code>check.cch</code> . The manifest constant for an enumerated type is <code>ENUM_TYPE</code> .

### 4.3.3 Bitfields

The incautious use of bitfields can lead to many portability problems. First, an obvious problem: compilers differ with respect to the maximum size of the bitfields that they allow. Thus a bitfield of 20 bits will compile without complaint on a 32-bit computer, but will cause a syntax error on a 16-bit computer.

Another portability problem can be caused by bitfields that are not explicitly declared `unsigned`. If such a bitfield is compiled by a compiler that considers bitfields to be unsigned, if the bitfield is initialized with a *negative* value, *e.g.* `-1` (a natural way to fill all the bits), and if the value of this field is then assigned to a signed `int` or `long`, then the result will be a small *positive* integer. However, a compiler that considers bitfields to be signed will obtain a negative result in the same assignment. Harbison & Steele (HS88:106) recommend that no type other than `unsigned` ever be used for a bitfield. They point out that a bitfield of type `int` can be compiled in three different ways: signed, unsigned, and pseudo-unsigned (see Glossary for definition). The signedness of a bitfield for any given C compiler is usually (but not necessarily!) the same as the signedness of a `char`.

Two additional minor portability problems: first, ANSI C compilers now allow bitfields to be members of unions, but most pre-ANSI compilers do not

allow this use of bitfields. Second, some compilers do not allow anonymous (unnamed) bitfields.

CodeCheck provides four variables for detecting these problems.

<b>Variable</b>	<b>Meaning</b>
<i>dcl_bitfield</i>	Set to 1 if a bitfield is found.
<i>dcl_bitfield_anon</i>	Set to 1 if an unnamed bitfield is found.
<i>dcl_bitfield_arith</i>	Set to 1 if a bitfield width requires calculation.
<i>dcl_bitfield_size</i>	Set to number of bits in a bitfield.
<i>dcl_union_bits</i>	Set to 1 if a bitfield is declared as a member of a union.

#### 4.3.4 Empty Declarations

Empty declarations are allowed in K&R C, but most are not permitted in ANSI C. The exception is an empty declaration in which the type specifier is a forward reference to a tag that will be needed later. A CodeCheck rule can be constructed to detect non-ANSI empty declarations, as follows:

```
1  if ( dcl_empty )
2  {
3      if ( (dcl_base != ENUM_TYPE)
4          && (dcl_base != UNION_TYPE)
5          && (dcl_base != STRUCT_TYPE) )
6          && (dcl_base != CLASS_TYPE) )
7      warn( 9999, "Empty declaration." );
8  }
```

The CodeCheck variables used in this rule are:

<b>Variable</b>	<b>Meaning</b>
<i>dcl_empty</i>	Set to 1 if an empty declarator is found.
<i>dcl_base</i>	Set to an integer which identifies the base type of the current declarator. The base types are defined as

manifest constants in the CodeCheck header file `check.cch`.

### 4.3.5 Microsoft “Based” Pointers

Microsoft C version 6.0 introduced into C a new and completely non-portable pointer type: the “based” pointer. To implement this new concept, four new non-portable keywords (`_segment`, `_segname`, `_based`, `_self`) and one new operator (`:>`) were introduced. In its default mode of operation CodeCheck will recognize these constructs, and will check for improper grammatical usage just as it does for the standard keywords and operators. To disable these and all other extended keywords and operators, specify `-K1` (for ANSI C) or `-K0` (for strict K&R C) on the command line.

### 4.3.6 Pascal Functions

Many implementations of C allow the programmer to call and write functions that use the conventions for passing arguments and returning values. Unfortunately, there is little agreement on how the `pascal` keyword should be used. Here are three representative cases:

- Microsoft C: `pascal` is a *type modifier*, similar to `cdecl`.
- MPW C: `pascal` is a *type specifier*, similar to `float`.
- Think C: `pascal` is a *storage class specifier*, similar to `static`.

As a result of these differences, it is difficult to use the `pascal` keyword in a fully portable way. One way to avoid the worst kind of trouble with this and similar keywords is never to declare more than one function per declarator list. Additionally, the CodeCheck function `keyword()` can be used to detect occurrences of this troublesome keyword.

```
1  if ( dcl_function && (dcl_count > 1) )
2      warn( 98, "Use only one declarator here." );
3
4  if ( keyword("pascal") )
```

```
5      warn( 99, "This keyword is not fully portable." );
```

<b>Variable/Function</b>	<b>Meaning</b>
<i>dcl_count</i>	Set to the index of the current declarator within a comma-delimited declarator list. The first declarator has index 1, the second 2, <i>etc</i> , until a semicolon is found that marks the end of the list.
<i>dcl_function</i>	Set to 1 if this is a function declaration.
<i>dcl_level()</i>	See Section 3.2 of the Reference Manual for details.
<i>keyword()</i>	Returns the integer value 1 whenever a keyword is found that matches the argument string.

### 4.3.7 Variable Numbers of Arguments

The ability to call functions with variable numbers and types of arguments is a powerful feature that differentiates C from most other high-level computer languages. Unfortunately, it also presents some portability problems. Some compilers, *e.g.* Microsoft C, allow users to indicate that a function takes a variable number of arguments by terminating the parameter list in the function declarator with a comma:

```
int main( argc, argv, )
```

This usage is not portable to ANSI standard C, nor to many other non-ANSI compilers. ANSI conforming compilers need to see an ellipsis (three dots ...) follow the last comma, while non-ANSI compilers generally consider the last comma to be a syntax error.

By the same token, the ANSI ellipsis notation for functions with variable numbers of arguments is equally non-portable to pre-ANSI compilers. CodeCheck provides three variables for handling these problems:

<b>Variable</b>	<b>Meaning</b>
<i>dcl_3dots</i>	Set to 1 whenever an ellipsis (...) is found.

<code>dcl_need_3dots</code>	Set to 1 when an ellipsis (...) is needed in a function parameter list, but is not found.
<code>dcl_oldstyle</code>	Set to 1 if an old-style ( <i>i.e.</i> not prototyped) function is declared.

### 4.3.8 Type Definitions

Some C compilers, *e.g.* MPW C, do not permit typedef names to be declared more than once within a module. Many compilers do tolerate this practice, however, so this becomes a portability problem for those programmers who acquire the dubious habit of placing identical type definitions within many header files.

*Recommendation:* Take care to define each type in exactly one header file, and use conditional compilation switches to guarantee that the contents of each header is compiled only once per module.

<b>Variable</b>	<b>Meaning</b>
<code>dcl_typedef_dup</code>	Set to 1 whenever a duplicate type definition is found.

### 4.3.9 Function typedef names

Many C compilers will accept typedef names for function types, and some will accept such names in function definitions. For example:

```

1  typedef void weird( int );
2  weird myfunc( int x )
3  {
4      /* What is the return type??? */
5  }
```

This usage is not permitted in ANSI C, because of the ambiguity involved in the return type of the function — is it a function returning *void*, or a function returning a function returning *void*? Compilers that permit this usage make the former assumption, because the latter is impossible in C.

*Recommendation:* Do not typedef function types, even if your compiler allows it. Here is a CodeCheck rule that will detect such type definitions:

```
1  if ( dcl_typedef )
2      if ( dcl_function )
3          warn( 99, "Do not typedef function types." );
```

<b>Variable</b>	<b>Meaning</b>
<i>dcl_function</i>	Set to 1 if this is a function declaration.
<i>dcl_typedef</i>	Set to 1 if a typedef name has been declared.

#### **4.3.10 Uninitialized static float or double variables**

The C language has always required that static variables without explicit initializers be initialized by the compiler to zero. However, a portability issue arises with static float, double, and long double variables: will the compiler use a zero bit-pattern for the initializer, or the floating-point representation of zero? The two are not necessarily the same. ANSI-conforming compilers are required to use the floating-point representation of zero, but many pre-ANSI compilers use the zero bit-pattern.

*Recommendation:* Always explicitly initialize static float, double, and long double variables. Here is a CodeCheck rule that will detect this problem:

```
1  if ( dcl_static )
2      if ( (dcl_simple || dcl_function )
3          && (! dcl_initializer)
4          && (dcl_base >= FLOAT_TYPE)
5          && (dcl_base <= LONG_DOUBLE_TYPE) )
6          warn( 99, "Need explicit initializer here." );
```

<b>Variable</b>	<b>Meaning</b>
<i>dcl_base</i>	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`.

<i>dcl_function</i>	Set to 1 if this is a function declaration.
<i>dcl_simple</i>	Set to 1 when a simple variable ( <i>i.e.</i> neither pointer, array, reference, nor function) is declared.
<i>dcl_initializer</i>	Set to 1 when an initializer is found.
<i>dcl_static</i>	Set to 1 when a non-local static identifier has been declared.

## 4.4 Portability at the Expression Level

### 4.4.1 Non-ANSI expressions

At least one popular compiler, Gnu C, allows parenthesized compound statements within expressions — an extension to C that is most certainly not part of the ANSI standard. For example, consider this macro defined in the Gnu header file `ctype.h`:

```
#define tolower(c) ( { int _c=(c); isupper(_c) ? _tolower(_c) : _c; } )
```

The purpose of this odd construction is to guarantee that the argument of the macro will be referenced only once. While this is a laudable goal, there is also the temptation for programmers to use this extension in their own code. This will create an immediate portability problem. CodeCheck has a trigger that will detect this (and other) non-ANSI expressions:

<b>Variable</b>	<b>Meaning</b>
<code>exp_not_ansi</code>	Set to 1 if a non-ANSI C expression is found. This variable does <i>not</i> trigger on C++ expressions that conform to the ANSI base document for C++.

## 4.5 Portability of Functions

### 4.5.1 Labels without Statements

Although no published grammar for C permits labels to exist without a statement that is to be labeled, many C compilers do permit this. For example, the label “end” in the following code fragment does not label any statement, even though it is clear what the author intended. This use of labels is forbidden in ANSI C, and should generate a syntax error when compiled by any strictly conforming compiler.

```
1     while ( x == 0 )
2     {
3         /* ... */
4     end:
5     }
```

*Recommendation:* For maximum portability, do not use labels without statements. The statement should be empty, *i.e.* a lone semicolon. CodeCheck has a predefined variable that will detect labels without statements:

<b>Variable</b>	<b>Meaning</b>
<code>stm_bad_label</code>	Set to 1 whenever a label or list of labels is found that is not attached to any statement.

### 4.5.2 Hidden Parameters

Traditional K&R compilers allow an identifier declared in the compound statement of a function (*i.e.* just after the first open brace) to have the same name as one of the function’s formal parameters, thus effectively hiding the parameter from the function. This practice is disallowed in the ANSI standard, and in the better H&S compilers. Here is an example of the problem:

```
1     void bomber( xptr )
2         char *xptr; /* The formal parameter is */
3         { /* hidden by the declared */
4         long *xptr; /* automatic variable. */
5
6         *xptr = 40000; /* This can crash the system, */
7         } /* but the function compiles. */
```

*Recommendation:* Since the only plausible circumstance in which this might occur is as a result of programmer error, there is little to say other than “don’t do it”. It may be an interesting test of your compiler to see if it can catch this bug. CodeCheck has a predefined variable that will find it:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>dcl_parm_hidden</i>	Set to 1 if a function parameter has the same name as an identifier declared within the function’s compound statement.

## 4.6 C Compiler Limits

### 4.6.1 Identifier, String, and Line Lengths

ANSI conforming C compilers allow internal identifiers to have a significance of at least 31 characters, but strict K&R compilers use a significance of only 8 characters. To further confuse the issue, the ANSI standard guarantees only 6 significant characters with no case sensitivity for external identifiers, due to the limitations of certain linkers. The variable `dcl_extern_ambig` should be used to detect portability problems caused by ambiguities in external identifiers, and the following rule may be used to enforce length limitations on internal identifiers:

```
1  if ( dcl_ident_length > 31 )
2      warn( 1001, "Identifier length exceeds 31." );
```

*Recommendation:* Use long names for both external and internal identifiers. If your linker demands short external names, then use the preprocessor to define your long names as short 6-character encoded names (as suggested in HS84:15). This will allow you to refer to variables by a long descriptive name, and still have short names for the linker. For example:

```
#define ptrRecentEventRecord    p32RER

/* ... intervening code ... */

extern EventRecord    *ptrRecentEventRecord;
```

ANSI conforming compilers allow string literals up to at least 509 characters in length, but the actual limits may vary widely. There is no guaranteed minimum among non-ANSI compilers. *Recommendation:* restricting string literals to 255 characters may be a safe, conservative policy.

```
1  if ( lex_str_length > 255 )
2      warn( 1002, "String too long for portability." );
```

C imposes no limit on line-length, but most compilers do have a limit, usually in the 100-500 character range. Lines longer than 80 characters may not be

portable. The POSIX.2 standard specifies a maximum line length of 2048 characters.

CodeCheck provides several predefined variables for detecting length violations:

<b>Variable</b>	<b>Meaning</b>
<i>dcl_extern</i>	Set to 1 if the <b>extern</b> storage class is explicitly specified in a declaration.
<i>dcl_extern_ambig</i>	If two external identifiers have names that agree on the first 6 or more characters, <i>regardless of case</i> , then this variable is set to the number of consecutive characters on which they agree.
<i>dcl_global</i>	Set to 1 if an identifier with external linkage has been declared. This includes variable, function, and typedef names.
<i>dcl_ident_length</i>	Set to the number of characters in the declared identifier.
<i>dcl_local</i>	Set to 1 if a local identifier has been declared.
<i>dcl_static</i>	Set to 1 when a non-local static identifier has been declared.
<i>lex_str_length</i>	Set to the length of a string literal (the terminating zero is not counted).
<i>lin_length</i>	Set to the number of characters in the line (excluding the newline character at the end of the line).

## 4.6.2 Preprocessor Limits

C compilers vary greatly with respect to limitations on the nesting of #if conditionals and #include directives. For example, ANSI conforming compilers must allow the nesting of #if conditionals to a depth of 8, but it is unsafe to assume that more than two or three will be portable (RJ89:195).

It is prudent to limit the depth of header file inclusion, because some preprocessors allow a nesting depth of only three or four.

The ANSI standard specifies that compilers must allow up to 31 formal parameters in macros, but pre-ANSI compilers may allow only four or five (RJ88:182).

CodeCheck provides three predefined variables for enforcing these preprocessor limits:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>pp_if_depth</i>	Set to the new depth of conditional compilation whenever an <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> , <code>#else</code> , <code>#elif</code> , or <code>#endif</code> directive is activated.
<i>pp_include_depth</i>	Set to the new depth of file inclusion whenever an <code>#include</code> directive is executed, or an end-of-file in a header file is encountered. See also <code>lin_source</code> .
<i>pp_arg_count</i>	Set to the number of formal parameters found in a macro definition.

# Chapter 5: Maintainable Style

Good C programming style is much more than simply indenting source code according to a specific formal system. Indeed, the indentation problem has become trivial as a result of the commercial publication of code “beautifiers”, which automatically adjust the indentation of a program to the specifications of the programmer. As many authors have pointed out, writing C with a good style requires adhering to a well thought-out list of rules for such things as choosing variable names, declaring variables, using preprocessor directives, *etc.* In many cases these rules can be expressed as CodeCheck rules.

This section describes how to use CodeCheck to monitor style for maintainability. The CodeCheck rules given here are based on various corporate standards for C coding that have been made available to Abraxas Software.

The C language is so large, its notation so compact, and its restrictions so few that stylistic conventions are absolutely mandatory for all C programmers. The only question is: how tough should they be? Most successful companies that employ teams of programmers use conventions and standards that are very tough indeed — and this toughness almost certainly contributes materially to their success. Discipline in itself does not inhibit creativity: this is just as true in the art of C programming as it is in any other art.

Because CodeCheck rules are written in a simplified form of C, the style rules can be edited and customized to accommodate almost any taste. A great number of CodeCheck variables have been predefined for an enormous range of stylistic issues, not just the ones identified by Thomas Plum. This means that project managers can design stylistic rule sets that correspond precisely to the issues that are of concern to them, and need not feel constrained to the style propounded by Plum.

## 5.1 Lexical Issues in Program Maintenance

### 5.1.1 Choosing Identifier Names

Programmers use a great variety of guidelines for choosing names for variables. One popular set of guidelines is paraphrased here.

1. Names should never be redefined in inner blocks. This practice confuses more than it helps, and should be avoided.
2. Function and typedef names must begin with a capital letter, variable names must begin with a lowercase letter.
3. Macro names must always be all uppercase, non-macro names must never be all uppercase.
4. Variable, function, and typedef names should never conflict with labels.
5. Class names must begin with a capital letter and has a corporate prefix, like Microsoft MFC class CEdit.
6. Data members of a class must have a prefix.

Some of the CodeCheck predefined variables and functions which can be used to detect violations of guidelines similar to the preceding are:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>dcl_all_upper</i>	Set to 1 if only uppercase letters are found in an identifier name when it is declared.
<i>dcl_enum_hidden</i>	Set to 1 when a declarator name hides an enumerated constant.
<i>dcl_first_upper</i>	Set to the number of initial uppercase letters in an identifier when it is declared.
<i>dcl_hidden</i>	Set to 1 if an inner-block declaration hides an outer.
<i>dcl_label_overload</i>	Set to 1 if an inner-block declarator name matches a label within the same function.

`dcl_typedef`

Set to 1 if a typedef name has been declared.

### 5.1.2 Hungarian Notation

Many companies have found that program maintenance can be simplified by adopting a spelling convention for identifiers which encodes the type of the variable in lowercase prefixes. The actual name of the identifier is distinguished from its prefixes simply by beginning the name with a capital letter. This is the so-called Hungarian notation.

Many variations on the Hungarian theme exist. One simple set of Hungarian prefixes is described below, together with a set of CodeCheck rules that test for compliance with these guidelines. This is an illustrative example only, and is not intended to be a complete specification for a Hungarian prefix scheme.

*Variable names must begin with one or more lowercase prefixes that describe the type of the variable. These prefixes are, in the order in which they must be used:*

1. "p" if this is a pointer, "np" if near pointer, "fp" if far pointer;
2. "a" if this is an array,
3. "c" if the base type is `char`, "s" if `short`, "i" if `int`, and "l" if `long`;
4. "g" if the base type is `float`, "d" if `double`;
5. "x" if the base type is `struct`, "w" if `union`, "en" if `enum`;
6. "b" if the base is the typedef name "Boolean".

For example, suppose that we want to declare that `Foo` is an array of far pointers to Booleans. The declared name of the identifier should be `afpbFoo` and the declaration should read:

```
Boolean far * afpbFoo[] = { /* initializers */ };
```

This spelling scheme can be enforced with one single CodeCheck rule:

```
1 #include <check.cch>
2
3 int k;
```

```

4
5 if ( dcl_global || dcl_static || dcl_local )
6 {
7     k = 0;
8     while ( k < dcl_levels )
9         {
10            switch ( dcl_level(k) )
11                {
12            case ARRAY:
13                if ( ! prefix("a") )
14                    warn( 1001, "a prefix missing on array." );
15                break;
16            case POINTER:
17                switch ( dcl_level_flags(k) )
18                    {
19            case NEAR_FLAG:
20                if ( ! prefix("n") )
21                    warn( 1002, "n prefix missing on near pointer." );
22                break;
23            case FAR_FLAG:
24                if ( ! prefix("f") )
25                    warn( 1001, "f prefix missing on far pointer." );
26                break;
27            }
28                if ( ! prefix("p") )
29                    warn( 1001, "p prefix missing on pointer." );
30                break;
31            }
32            k++;
33        }
34    switch ( dcl_base )
35        {
36    case VOID_TYPE:
37        if ( ! prefix("v") )
38            warn( 1001, "v prefix missing on void." );
39        break;
40    case ENUM_TYPE:
41        if ( ! prefix("n") )
42            warn( 1001, "v prefix missing on enum." );
43        break;
44    case CHAR_TYPE:
45        if ( ! prefix("c") )
46            warn( 1001, "c prefix missing on char." );
47        break;
48    case SHORT_TYPE:
49        if ( ! prefix("s") )
50            warn( 1001, "s prefix missing on short." );
51        break;
52    case INT_TYPE:
53        if ( ! prefix("i") )
54            warn( 1001, "i prefix missing on int." );
55        break;
56    case LONG_TYPE:

```

```

57     if ( ! prefix("l") )
58         warn( 1001, "l prefix missing on long." );
59     break;
60 case FLOAT_TYPE:
61     if ( ! prefix("g") )
62         warn( 1001, "g prefix missing on float." );
63     break;
64 case DOUBLE_TYPE:
65     if ( ! prefix("d") )
66         warn( 1001, "d prefix missing on double." );
67     break;
68 case UNION_TYPE:
69     if ( ! prefix("w") )
70         warn( 1001, "w prefix missing on union." );
71     break;
72 case STRUCT_TYPE:
73     if ( ! prefix("x") )
74         warn( 1001, "x prefix missing on struct." );
75     break;
76 case DEFINED_TYPE:
77     if ( strcmp(dcl_base_name(),"Boolean") == 0 )
78         if ( ! prefix("b") )
79             warn( 1016, "b prefix needed on %s.", dcl_name() );
80     break;
81     }
82 }

```

Some of the CodeCheck predefined variables and functions which can be used to detect violations of guidelines similar to the preceding are listed below. The full list of declarator variables is given in the CodeCheck Reference Manual.

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>dcl_base</i>	Set to a constant that identifies the base type of the declarator (see Reference Manual section 3.2 for details).
<i>dcl_base_root</i>	Type from which the type of <i>dcl_base</i> is derived. If the type of <i>dcl_base</i> is not a user-defined type, <i>dcl_base_root</i> has same value as <i>dcl_base</i> .
<i>dcl_base_name()</i>	Returns the name of the base type of the declarator.
<i>dcl_base_name_root()</i>	The name of type from which type of <i>dcl_base_name</i> is derived. If the type of <i>dcl_base_name</i> is not a user-defined type, <i>dcl_base_name_root()</i> returns the same value as <i>dcl_base_name()</i> .

<i>dcl_explicit</i>	Set to 1 when a declarator has specifier "explicit".
<i>dcl_extern</i>	Set to 1 if the <code>extern</code> storage class is explicitly used in a declaration.
<i>dcl_from_macro</i>	Set to 1 when declarator name is derived from a macro expansion.
<i>dcl_global</i>	Set to 1 if an identifier with external linkage has been declared. This includes variable, function, and typedef names.
<i>dcl_hidden</i>	Set to 1 if an inner-block declaration hides an outer.
<i>dcl_Hungarian</i>	Set to 1 if the Hungarian style is detected (a capital letter is immediately preceded by a lowercase letter).
<i>dcl_levels</i>	Number of levels in the type of this declarator.
<i>dcl_local</i>	Set to 1 if a local identifier has been declared.
<i>dcl_member</i>	1 when a union member identifier is declared, 2 when a struct member identifier is declared, 3 when a class member identifier is declared;  ( C++ members may be: variables, functions, or typedef names ).
<i>dcl_mutable</i>	1 when an identifier is declared 'mutable'.
<i>dcl_scope_name( )</i>	The scope name of current declarator.
<i>dcl_signed</i>	Set to 1 if the <code>signed</code> type specifier is explicitly used in a declaration.
<i>dcl_static</i>	Set to 1 when a non-local static identifier has been declared.
<i>dcl_typedef</i>	Set to 1 if a typedef name has been declared.
<i>dcl_unsigned</i>	Set to 1 when the type specifier <code>unsigned</code> is used in a declaration.

<i>op_declarator</i>	Any operator found within a declaration.
<i>prefix()</i>	Set to 1 if the next prefix in the declarator name matches the argument string.
<i>suffix()</i>	Set to 1 if the next suffix in the declarator name matches the argument string.

### 5.1.3 Manifest Constants

Plum recommends several guidelines for defining manifest constants (TP84:18-19). A constant is “manifest” if its meaning is clearly apparent to the maintenance programmer. Manifest constants are used to give meaning to numbers which otherwise would seem wholly capricious, as may be seen in these two contrasting examples:

#### Example 1 — bad, 32 has no manifest meaning:

```
1  if ( index < 32 )
2      x[index++] = 0;
```

#### Example 2 — good, 32 now has a clear meaning:

```
1  #define  TABLSIZE  32
2      ...
3      ...
4  if ( index < TABLSIZE )
5      x[index++] = 0;
```

As Plum points out, manifest constants are neither manifest nor constant if their value is changed midway through a program (by means of an `#undef` and/or another `#define` that redefines the constant). To change a manifest constant in this manner is to create an immediate maintenance problem. Indeed, there only three valid purposes for an `#undef`:

1. to override a macro name defined in a standard header,
2. to limit the lexical scope of a macro,
3. to free up space in the compiler’s macro table, if it threatens to overflow.

The Plum guidelines for manifest constants that can be checked by CodeCheck are paraphrased as follows:

- a. The names of manifest constants should be spelled in capital letters.
- b. The value of manifest constants must never change.
- c. If a manifest constant is to be used in more than one file, then it must be defined in a single header file.

The predefined CodeCheck variables which are used to detect violations of these guidelines are:

<b>Variable</b>	<b>Meaning</b>
<i>lex_not_manifest</i>	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.
<i>lex_initializer</i>	Set to the following when an initializer is found: 1 if the initializer is the integer zero, 2 if the initializer is a non zero integer, 3 if initializer is a boolean literal, true or false(C++ only). 4 if the initializer is a float or double constant, 5 if the initializer is a string literal, and 6 if the initializer is anything else.
<i>pp_lowercase</i>	Set to 1 if the macro name in a macro definition is defined with any letters that are lowercase.
<i>pp_stack</i>	Set to 1 if a macro is multiply defined.
<i>pp_undef</i>	Set to 1 whenever #undef is used.
<i>pp_macro_dup</i>	Set to 1 if a macro is defined in more than one file.
<i>pp_macro_conflict</i>	Set to 1 if a macro is defined differently in more than one file.

### 5.1.4 Lexical Rules for Spacing Operators

C code readability is greatly enhanced by rigorous adherence to a uniform system for placing spaces around C operators and punctuation marks. Elaborating somewhat upon the spacing scheme recommended by Plum (TP84:30-32), CodeCheck recognizes six categories of tokens that need spacing:

1. *High precedence operators*: all unary operators, and the four member selection operators ( . -> .\* ->\* ). These tokens should never have space separating them from their operands.
2. *Low precedence operators*: all assignment operators and the conditional operator pair ( ? : ). These tokens should always have space separating them from their operands.
3. *Commas, semicolons, and colons*: these tokens should not be preceded by a space, and should be followed by a space or newline.
4. *Function argument parentheses*: the open parenthesis in a function call should not be separated by whitespace from the function identifier. There should be consistency within a project as to whether or not the contents of the parenthetical expression are separated by spaces from the parentheses.
5. *Subscript selection brackets*: the open bracket should not be separated by whitespace from the array to which it refers. There should be consistency within a project as to whether or not the contents of the bracket expression are separated by spaces from the brackets.
6. *Other*: This category encompasses all other operators not listed above, including arithmetic, bitwise, relational, and logical operators. These tokens are usually but not necessarily surrounded by spaces.

The predefined CodeCheck variables with which these stylistic guidelines may be checked are given below. In designing a set of lexical rules relating to spacing, the goal should be to enforce consistency to a simple and clearly-stated spacing scheme.

<b>Variable</b>	<b>Meaning</b>
<code>lex_punct_after</code>	Set to 1 if a comma or semicolon is not followed by a whitespace character, a comma, or a semicolon.

<i>lex_punct_before</i>	Set to 1 if a comma or semicolon is preceded by a space.
<i>lin_continues</i>	Set to 1 if a line ends before the end of the current expression.
<i>op_executable</i>	Set to 1 if the operator is within executable code, including initializers.
<i>op_declarator</i>	Set to 1 if the operator is within a declaration, <i>not</i> including initializers.
<i>op_low</i>	Set to 1 for operators with low precedence.
<i>op_medium</i>	Set to 1 for operators with medium precedence.
<i>op_high</i>	Set to 1 for operators with high precedence.
<i>op_prefix</i>	Set to 1 for unary prefix operators.
<i>op_infix</i>	Set to 1 for binary infix operators.
<i>op_postfix</i>	Set to 1 for unary postfix operators.
<i>op_space_before</i>	Set to 1 if an operator or punctuation mark is preceded by a space character.
<i>op_space_after</i>	Set to 1 if an operator or punctuation mark is followed by a space character.
<i>op_white_before</i>	Set to 1 if an operator or punctuation mark is preceded by whitespace.
<i>op_white_after</i>	Set to 1 if an operator or punctuation mark is followed by whitespace.

### **5.1.5 Indentation and the Placement of Braces**

The purpose of indentation is to reveal the subordinate nature of blocks of code, thus greatly enhancing program readability. Plum has summarized the

fundamental rule for indentation which underlies almost all popular formats (TP84:42):

*Each line which is part of the body of a C control structure (if, while, do-while, for, switch) is indented one tab stop from the margin of its controlling line. The same rule applies to function, struct, or union definitions, and aggregate initializers.*

CodeCheck provides several predefined variables that allow indentation to be checked, using Plum's general guideline as a basis for individualized schemes.

The fundamental rule stated above leaves unspecified how braces are to be treated. Oddly enough, the opinions of C programmers on this point seem to be held with a fervor approaching religious fanaticism. CodeCheck resolves the problem by providing a command-line option (**-B**) which, if set, informs CodeCheck that braces are to be considered part of the body of the control structure (this corresponds to Plum's own preference, and to the convention used in Pascal). If **-B** is not set, then braces are not so considered (the Kernighan & Ritchie style). The **-B** option only affects the way in which the predefined CodeCheck variable `lin_nest_level` is incremented and decremented. Two examples may serve to illustrate this difference:

<u>Option -B not set (default)</u>		<u>lin_nest_level</u>		
1	if ( test > 0 )	/*	1	*/
2	{	/*	1: no indent	*/
3	k += delta;	/*	2	*/
4	printf( "%d\n", k );	/*	2	*/
5	}	/*	1: no indent	*/
6	printf( "Done" );	/*	1	*/

<u>Option -B set</u>		<u>lin_nest_level</u>		
1	if ( test > 0 )	/*	1	*/
2	{	/*	2: indented brace	*/
3	k += delta;	/*	2	*/
4	printf( "%d\n", k );	/*	2	*/
5	}	/*	2: indented brace	*/
6	printf( "Done" );	/*	1	*/

The CodeCheck predefined variable `lin_nest_level` not only counts the depth of control structures, it is also incremented and decremented appropriately within function definitions, structure and union definitions, and aggregate initializers.

<b>Variable</b>	<b>Meaning</b>
<i>lin_nest_level</i>	Set to the nesting level when the first nonwhite character of a line is found. How braces are counted in the nesting level depends on the command line option – <b>B</b> .
<i>lin_indent_space</i>	Set to the number of leading space characters found before the first non-white non-comment character of a line.
<i>lin_indent_tab</i>	Set to the number of leading <i>tab</i> characters found before the first non-white non-comment character of a line.
<i>lin_continuation</i>	Set to 1 if a line continues an expression or declaration list from the previous line.

Some standards ask programmers to use only space characters to indent, while others specify only tab characters. In these cases the CodeCheck variables *lin\_indent\_space* and *lin\_indent\_tab* can be used to determine the amount of indentation and to detect the use of the wrong character to create indentation.

Determining the actual width of the indentation used on any given line of code is problematic, unless the characteristics of the editor used to display the code are known. Some editors (e.g. Macintosh and Windows editors) use proportional-width characters, while older editors use single-width characters. Worse, editors vary greatly in how they treat tab characters: primitive editors simply generate 8 spaces when a tab is entered, better editors move forward to a fixed position on the page, while yet others behave in a context-dependent way. In the face of this chaos, it is simply not possible to find a general formula that calculates the actual displayed indentation given a sequence of mixed spaces and tabs.

Here is an example CodeCheck rule that checks indentation within function definitions, based on the number of leading tab characters in each line. This rule ignores the indentation of comments, preprocessor directives, and declarations.

```

1  int  difference;
2
3  if ( lin_within_function )
4      if ( ! ( lin_is_comment || lin_preprocessor || lin_dcl_count ) )
5          {
6              difference = lin_nest_level - lin_indent_tab;
7              switch ( difference )
8                  {
9                  case -1:
10                 warn( 1003, "Indent 1 fewer tab." );
11                 break;
12                 case 0:      // Indentation is correct.
13                 break;
14                 case 1:
15                 warn( 1003, "Indent 1 more tab." );
16                 break;
17                 default:
18                 warn( 1004, "Indent %d tabs.", difference );
19                 break;
20             }

```

Most standards for the indentation of switch statements call for an appearance like this:

```

1      switch ( ijk )
2      {
3      case 1:
4          break;
5      default:
6          break;
7      }

```

However, some standards call for more indentation of the contents of the switch, as in this example:

```

1      switch ( ijk )
2      {
3          case 1:
4              break;
5          default:
6              break;
7      }

```

CodeCheck assumes that the former example is the norm. To enforce a standard that calls for more indentation, like the latter example, the following rule can be used:

```

1  #include <check.cch>
2
3  int  start_switch, // Flags beginning of a switch statement.
4      switch_depth, // Measures the depth of switch nesting.

```

```

5      diff;          // Amount of error in nesting (in tabs).
6
7  if ( stm_cp_begin == SWITCH )
8      start_switch = 1;          // Detect start of switch
9
10 if ( stm_is_comp == SWITCH )
11     --switch_depth;          // Detect end of switch
12
13 if ( lin_within_function )
14     if ( ! (lin_is_comment || lin_preprocessor || lin_dcl_count) )
15         {
16             diff = lin_nest_level + switch_depth - lin_indent_tab;
17             switch ( diff )
18                 {
19                 case -1:
20                     warn( 1003, "Indent 1 fewer tab." );
21                     break;
22                 case 0:      // Indentation is correct.
23                     break;
24                 case 1:
25                     warn( 1003, "Indent 1 more tab." );
26                     break;
27                 default:
28                     warn( 1004, "Indent %d tabs.", difference );
29                     break;
30                 }
31             if ( start_switch )
32                 {
33                     ++switch_depth;
34                     start_switch = 0;
35                 }
36         }

```

The variables used in this example are:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>stm_cp_begin</i>	When the open curly brace of a compound statement has been found, this variable is set to the context of the compound statement (see Reference Manual section 3.12 for details).
<i>stm_is_comp</i>	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).

Some standards require that if, else, for, while and do statements contains compound statements even though the compound statements themselves contain only single statements or empty statements.

The variable for this purpose is

<b>Variable</b>	<b>Meaning</b>
<i>stm_need_comp</i>	Set to 1 if a statement contained by if, else, while, do and for is not a compound statement.

### 5.1.6 Postfix Problems

For reasons that are impossible to understand, C has always allowed a *long* numeric constant to be indicated with a postfix *lowercase* 'el', as in the constant pi: 3.14159265351. It takes great concentration to discern that the last character in this constant is an 'el' and not a 'one'.

As a matter of style, some project leaders do not permit the use of suffixes at all, preferring to see explicit casts. The predefined CodeCheck variable which will detect any suffix attached to a numeric constant is *lex\_suffix*.

*Recommendation:* Avoid the lowercase 'el'. Always use an explicit cast or the uppercase 'L' postfix to indicate a *long*.

<b>Variable</b>	<b>Meaning</b>
<i>lex_lc_long</i>	Set to 1 if a numeric constant ends with a lower-case 'el', indicating a <i>long</i> type.
<i>lex_suffix</i>	Set to 1 if a numeric constant is found with any suffix ('F', 'f', 'L', 'l', 'U', or 'u'), or combination of these.

### 5.1.7 Nonstandard Comments

A small minority of C compilers allow the nesting of comments within comments. This usage is therefore not portable. CodeCheck can recognize both kinds of nonstandard comments, but treats a nested */\* ... \*/* comment as a syntax error. To avoid the syntax error, set the *-N* option on the command-line that invokes CodeCheck, or have a rule that calls *set\_option('N', 1)* at the start of each module.

If you want CodeCheck to decide that nested comments are acceptable as soon as the first nested comment is found, use this rule:

```
1  if ( lin_nested_comment )
2    {
3    warn( 1234, "Nested comment." );
4    set_option( 'N', 1 );
5    }
```

Some compilers have adopted the `//` comment of C++. These comments are terminated by the end of the line on which the `//` was found. Needless to say, these comments are not portable either. CodeCheck accepts a `//` comment as syntactically correct, and excepts without complaint the nesting of `//` comments within `/* ... */` comments and *vice versa*.

*Recommendations:* (a) To block out a section of code that contains comments, bracket the section with `#if 0` at the top and `#endif` at the bottom. This will serve to disable the entire section of code, even if it contains comments. (b) Do not use the `//` comment format unless you are actually writing C++ code.

CodeCheck provides two predefined variables for detecting nonstandard comments:

<b>Variable</b>	<b>Meaning</b>
<code>lin_nested_comment</code>	Set to 1 if a <code>/*...*/</code> comment is found nested within another <code>/*...*/</code> comment.
<code>lex_cpp_comment</code>	Set to 1 if a <code>//</code> comment is found.

### 5.1.8 Magic Numbers

A “magic number” is a number whose meaning is not apparent to the maintenance programmer. The example in section 5.1.3 illustrates the use of a magic number, in this case 32. There are three problems with magic numbers:

1. The meaning of a simple number is almost never apparent from context alone.

2. It is very difficult to change all occurrences of a particular magic number when its value must be changed, because not all occurrences of a particular number will have the same meaning.
3. A magic number adds to a program's cognitive complexity because it forces the reader to wonder what the meaning of the number might be.

*Recommendation:* Either define constants other than 0 and 1 as manifest constant macros (e.g. `#define buffno 15`) or use a constant declaration with an initializer (e.g. `const short buffno = 15;`). If you use a macro for the constant, then place its definition in a single header file. Here is a CodeCheck rule for detecting magic constants:

```
1  if ( lex_not_manifest )
2      if ( lex_initializer < 5 && lex_initializer != 3 )
3          warn( 99, "Use a const or macro for this magic number!" );
```

### **Variable**

### **Meaning**

<i>lex_initializer</i>	Set to the following when an initializer is found: 1 if the initializer is the integer zero, 2 if the initializer is a non zero integer, 3 if initializer is a boolean literal, true or false ( C++ only ), 4 if the initializer is a float or double constant, 5 if the initializer is a string literal, and 6 if the initializer is anything else.
<i>lex_not_manifest</i>	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.

### 5.1.9 Escape Sequences in Character Constants and String Literals

An escape sequence within a character constant or string literal is signaled by the backslash character: \. The unrestricted use of escape sequences is a frequent cause of maintenance problems.

*Recommendation:* for maximum program clarity and maintainability, observe these rules:

1. To enhance program readability, use macros to encode character constants that contain escape sequences, as in `#define DBLQUOTE '\\"'`
2. Avoid numeric escape codes if possible; if unavoidable then use a macro definition to encode the constant, so that the purpose is evident in the name of the macro.

CodeCheck provides a predefined variable for detecting numeric escapes:

<b>Variable</b>	<b>Meaning</b>
<code>lex_num_escape</code>	Whenever a non zero numeric escape sequence is found, the value of this variable is set to the value of the numeric escape sequence.

## 5.2 Preprocessor Considerations

### 5.2.1 Silent Preprocessor Errors

The preprocessor permits programmers to make a variety of mistakes without complaint. These constitute maintenance problems as well, because programs containing these errors may appear to compile and execute correctly.

First, many compilers ignore code that appears on the same line as an `#ifdef` directive. In the following example the desired debug message will never be printed:

```
1  #ifdef DEBUG printf( "Now entering recursion..." );
2  #endif
```

Second, even experienced programmers may sometimes place an assignment operator in a simple substitution macro, as in the following code (which will compile, although not the way the programmer intended!).

```
3  #define TOOBIG = 99999
```

This is a case in which the permissive grammar of the preprocessor actually encourages silent but deadly program errors. A similar error occurs when a programmer inadvertently concludes a macro definition with a semicolon.

Third, some programmers forget (or do not realize) that it is important always to surround macro formal parameters with parentheses.

There are four predefined CodeCheck variables for detecting these common preprocessor errors:

<b>Variable</b>	<b>Meaning</b>
<i>pp_trailer</i>	Set to 1 if a preprocessor line contains any nonwhite characters after the end of the directive and before the end of the line.
<i>pp_assign</i>	Set to 1 if a macro definition is a simple assignment.
<i>pp_semicolon</i>	Set to 1 if a macro definition ends with a semicolon.
<i>pp_arg_paren</i>	Set to 1 if a macro formal parameter is used without being surrounded by parentheses.

## 5.2.2 The #define/#undef Morass

Any constant which might need to be changed during program development or maintenance should be “manifest”, which means that it must be *clearly apparent to the sight or understanding* (TP84:18). Manifest constants are used to give meaning to numbers which otherwise would seem wholly capricious, as may be seen in the two contrasting examples in section 5.1.3.

A problem is caused by programmers who have learned that their compiler treats the #define directives as though they were pushing definitions onto a stack, and #undef directives as though they were popping these definitions off the stack. Using #define and #undef in this way is not only an abuse of the preprocessor, it is also highly non portable. Not all preprocessors are set up in this way, and the ANSI standard specifically forbids this behavior. The CodeCheck preprocessor adheres to the ANSI standard in this matter, and flags every apparent pop of the macro stack by setting the predefined variable `pp_unstack` (defined below).

There is another way in which #undef can cause tearing of hair and gnashing of teeth, as illustrated in this highly artificial example:

```
1  #define  BLEEP  3
2  #define  BLATZ  BLEEP + BLEEP
3  #undef   BLEEP
```

What will your compiler see if you now refer to `BLATZ`? The ANSI standard is quite clear that `BLATZ` must be replaced by `BLEEP+BLEEP`, but some pre-ANSI compilers will receive `3+3` from their preprocessors. CodeCheck flags this kind of #undef usage by setting the predefined variable `pp_depend`.

Lastly, there is the ANSI concept of “benign redefinition”. ANSI has blessed the idea that macros can be multiply defined (*e.g.* in several different header files) as long as the definitions are virtually identical. Two macro definitions are virtually identical if they are exactly identical after each whitespace sequence has been replaced by a single space character. Any other redefinition of a macro will be flagged by an ANSI preprocessor as an error. The CodeCheck preprocessor allows benign redefinition, but sets the variable `pp_benign` whenever a benign redefinition is encountered.

<b>Variable</b>	<b>Meaning</b>
<i>pp_benign</i>	Set to 1 if a macro is redefined to be virtually identical to its previous definition.
<i>pp_depend</i>	Set to 1 if <code>#undef</code> is used on a macro that is used by other macros.
<i>pp_stack</i>	Set to 1 if a macro is redefined within a module (0 if the redefinition is benign).
<i>pp_undef</i>	Set to 1 whenever <code>#undef</code> is used.
<i>pp_unstack</i>	Set to 1 if <code>#undef</code> is used to unstack multiply-defined macros.

*Recommendation:* define your manifest constants once in a header file, and leave them defined. Do not depend upon benign redefinition, and do not use `#undef` unless absolutely required by overflow of the macro symbol table.

## 5.3 Maintainability in Declarations

### 5.3.1 Identifier Name Length

For program clarity and maintainability, it is desirable to use identifier names that are long enough to express the purpose to which they will be put. Thus identifier length can and should be used as one indicator of maintainability.

*Recommendation:* Give each variable a name that succinctly describes its purpose.

<b>Variable</b>	<b>Meaning</b>
<code>dcl_ident_length</code>	Set to the number of characters in the declared identifier.

Here is a sample CodeCheck program for measuring the average identifier name length within each module of a project, and flagging any identifiers that are over 31 or under 3 characters in length. The assignment in line 9 is necessary because `dcl_ident_length` is not itself a statistic.

```
1  statistic float   length;
2
3  if ( dcl_ident_length )
4  {
5      if ( dcl_ident_length > 31 )
6          warn(5000, "Identifier exceeds 31 characters!");
7      if ( dcl_ident_length < 3 )
8          warn(5001, "Use a longer name (>3 characters).");
9      length = dcl_ident_length; // save current length
10 }
11
12 if ( mod_begin )
13     reset(length);
14
15 if ( mod_end )
16 {
17     meanIDLength = mean( length );
18     printf("Mean identifier length = %g", meanIDLength);
19 }
```

### 5.3.2 Declaration Format

Plum (TP84:2) and many others strongly recommend some fairly stringent guidelines for formatting declarations. The guidelines reported here are those intended to improve the maintainability of C programs.

1. All variables, functions, and function parameters must have an explicit type specifier.
2. Declare only one variable, function, tag or member per line of source code.
3. Place an explanatory comment at the end of every declaration line.

The predefined CodeCheck variables which are used to detect violations of the above guidelines are:

<b><i>Variable</i></b>	<b><i>Meaning</i></b>
<i>dcl_no_specifier</i>	Set to 1 if a variable or function declarator has no explicit type information.
<i>dcl_not_declared</i>	Set to 1 if an old-style function parameter is not declared.
<i>dcl_parameter</i>	Index of function parameter (1 for first, etc.)
<i>lin_dcl_count</i>	Set to the number of identifiers declared on the current line (includes tag definitions and function parameters).
<i>lin_has_comment</i>	Set to 1 if a line has a comment that contains text.

In this example we restrict one variable, function declaration per line, it is unnecessary to count function parameters. The following rule will exclude function parameters.

```
int parmCount;
if ( mod_begin ) paramCount = 0;           // reset
if ( dcl_parameter ) paramCount++;
if ( lin_end ) {

    if ( lin_dcl_count - paramCount > 1 ) {
```

```

        warn( 99, "Only use one declarator per line." );
    }
    paramCount = 0;           // reset count at EOL
}

```

### 5.3.3 Initialization of External Variables

The ambiguity in C between *defining* and *referencing* external declarations makes possible an extremely bizarre class of bugs, some of which do not manifest themselves until an apparently stable program is substantially revised. From a programmer's point of view, the best defense against these bugs is strict adherence to two guidelines, paraphrased from Harbison & Steele (HS87:80):

1. Define each external variable in only one source file. Indicate that this is a definition by omitting the `extern` storage class keyword from the declaration, and supplying an initializer.
2. Every non-defining declaration of an external variable must use the `extern` storage class keyword, and must not have an initializer.

These two rules appear to be the only rules that will simultaneously protect programmers from bugs introduced by missing or conflicting initializers, and also ensure portability to non-ANSI compilers that use idiosyncratic methods for distinguishing defining from referencing declarations. Here is a CodeCheck rule that checks for adherence to these guidelines:

```

1  if ( dcl_variable )
2      if ( dcl_global )
3          {
4              if ( dcl_extern && dcl_initializer )
5                  warn( 8001, "Initializer must be omitted." );
6              else if ( (! dcl_extern) && (! dcl_initializer) )
7                  warn( 8002, "Initializer needed here." );
8          }

```

<b>Variable</b>	<b>Meaning</b>
<code>dcl_extern</code>	Set to 1 if the <code>extern</code> storage class is explicitly specified in a declaration.
<code>dcl_global</code>	Set to 1 if an identifier with external linkage has been declared. This includes variable, function, and type-def names.

<code>dcl_initializer</code>	Set to 1 when an initializer is found.
<code>dcl_template</code>	Number of C++ function template parameters.
<code>dcl_variable</code>	Set to 1 if a variable is declared.

### 5.3.4 Keywords `const`, `and volatile` as Type Modifiers

Many C compilers for DOS and OS/2 (e.g. Microsoft, Borland and Intel) have special keywords that modify the type of a declarator. These keywords include `near`, `far`, `cdecl`, `pascal`, and others. Programmers must be alert to the fact that these special non-ANSI type modifiers do not act like ordinary type specifiers: *they only modify the type of the next declarator or pointer, and have no effect on any other declarator in the list.* For example, in this declaration list

```
short far p, q;
```

the declarator `p` is a far short integer, but `q` is only a plain short integer. (*Except when compiled by Metaware High C, for which `near`, `far`, and `huge` are type specifiers. The possibilities for confusion are truly endless.*)

A serious ambiguity arises with `const` and `volatile`: some compilers accept these keywords both as ANSI standard type qualifiers and as non-ANSI type modifiers, depending on context. Consider these four declaration lists:

```
1  const short      a, b;          /* ANSI, both are constant */
2  short const     c, d;          /* ANSI, both are constant */
3  short far const e, f;          /* Not ANSI, f is NOT constant */
4  short           g, const h;    /* Not ANSI, h is constant */
```

In the first two declaration lists `const` is used as a type *specifier*, which applies to every variable in the declaration list. The third declaration list illustrates a non-ANSI use of `const` that is allowed by the Microsoft C compiler. Here the keyword `const` is a type *modifier* (modifying only the next declarator), hence only `e` is a far constant short integer, while `f` is a plain short integer. The fourth declaration list illustrates another non-ANSI use of `const` that is allowed by Microsoft and Intel compilers; in this list only `h` is constant.

Here is a simple rule that will detect these potentially troublesome usages:

```
1  if ( dcl_cv_modifier )
2      warn( 8026, "Non-ANSI usage of const or volatile." );
```

<b>Variable</b>	<b>Meaning</b>
<i>dcl_cv_modifier</i>	Set to 1 if the keyword <code>const</code> is used as a non-ANSI type modifier (similar to <code>near</code> , <code>far</code> , <i>etc.</i> ) rather than as an ANSI type specifier. Set to 2 if the keyword <code>volatile</code> is used as a non-ANSI type modifier.

C++ has the following special keywords .

<i>dcl_explicit</i>	1 when a declarator has specifier <i>explicit</i> .
<i>dcl_mutable</i>	1 when an indentifier is declared <i>mutable</i> .

## 5.4 Maintainability at the Project Level

### 5.4.1 Macro Redefinition

Macros that are defined differently within the several files of a project are at best a source of great confusion for the maintenance programmer, and at worst can cause some of the most mysterious program behaviors ever encountered.

*Recommendation:* If a macro is used in more than one source file of a project, then place its definition in a single header file. Each macro should have exactly one definition per project. Be sure to document the definition with a comment.

CodeCheck normally compares each macro definition with every other macro of the same name defined in other modules. If the current definition differs in any significant way (*i.e.* it is not a “benign redefinition” in ANSI terminology), then CodeCheck emits the warning message C0008 (see Reference Manual, Section 5.2). The CodeCheck variable `pp_macro_conflict` is also set at this time, so that users can trigger their own customized error messages.

CodeCheck provides three predefined variables for working with macro redefinition problems:

<b>Variable</b>	<b>Meaning</b>
<code>pp_macro_dup</code>	Set to 1 if a macro is defined in more than one file.
<code>pp_macro_conflict</code>	Set to 1 if a macro is defined differently in separate modules of a project.
<code>prj_conflicts</code>	Set to the number of conflicting macro definitions found in a project.

# Chapter 6: Software Metrics

Good management of the production and maintenance of computer software is at best very difficult. A major part of the problem is caused by difficulties in measuring such crucial quantities as programmer productivity, quality of code, and defect rate. Programmer productivity, for example, is often measured in thousands of lines of code per month, as though all programmers wrote lines in roughly the same way. Unfortunately for this measure, some write with great expanses of whitespace, blank lines, and comments, while others produce dense agglomerations of turgid code. Line for line, the functional capability of the dense code may be ten times that of the expansive code. Should the programmer who writes dense code be penalized for this? On the other hand, the expansive code is almost always easier to maintain, because it is more comprehensible to the maintenance programmer, so clearly there are trade-offs that must be quantified.

The emerging discipline of software engineering has produced many experimental metrics which attempt to quantify various aspects of computer code. The literature on software metrics is still quite sparse compared to the more mature engineering disciplines, but many good ideas have appeared. A good textbook on software measurement was finally published in 1986: **Software Engineering Metrics and Models**, by Conte, Dunsmore, and Shen. In addition, Grady and Caswell have written an excellent case history of Hewlett-Packard's efforts in software measurement, entitled **Software Metrics: Establishing a Company-Wide Program**.

## 6.1 Program Size

Measures of program size are fundamental to the management of software engineering, because they provide the units on which many important measures are based. Productivity, for example, is often expressed in thousand lines of code per programmer per month, and accuracy can be measured in defects per hundred lines of code.

Software Engineers have attempted to resolve the problems of the "lines of code" measure in two ways: (1) tinkering with the definition of a line of code, so

that it (for example) excludes blank and comment lines, as discussed in section 6.1.2, and (2) defining other measures of program size, as discussed in section 6.1.3 (statements), 6.1.4 (tokens) and 6.1.5 (functions). There is, however, a third way to deal with this problem: use distributions.

### **6.1.1 Distributions Provide Additional Information**

An approach to the problem of defining program size which is both more meaningful and appropriate from a statistical point of view, is to focus on *the distribution of kinds of lines of code*. This approach has not yet appeared in the software engineering literature, even though it is common enough in the context of general measurement tools for business management. As an example of this new focus, one could define the optimum distribution of kinds of lines of code as 10% blank, 30% comment, 30% declarations, and 30% executable code. Now a program can be described in two complementary ways: total lines and deviation from the optimum mix of kinds of lines. All of the measures of program size that can be calculated by CodeCheck can be considered in this manner, by declaring the CodeCheck variable for the measure to be of the `Statistic` storage class.

### **6.1.2 Lines of Code**

Although there is no consensus within software engineering as to how exactly to define a line of code, the actual variations concern how to treat: (1) blank lines, (2) comment lines, (3) lines derived from header files, (4) lines suppressed by the preprocessor, (5) multiple and/or fragmentary statements, and (6) non-executable statements. One popular and useful definition (CDS86:35) reads as follows:

*A **line of code** is any line of program text that is not a comment or a blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.*

The above definition is silent with regard to whether lines are to be counted if they are suppressed or if they come from header files, and these are points on which reasonable men and women can and do differ.

CodeCheck is flexible enough to handle most variations of the measure for lines of code. There are 10 predefined variables that describe features of each line:

**Line variables:**

<i>Variable</i>	<i>Meaning</i>
<i>lin_end</i>	Set to 1 when an end-of-line marker has been found (a newline character or the backslash-newline pair).
<i>lin_dcl_count</i>	Set to the number of identifiers declared on the current line (includes tag definitions and function parameters).
<i>lin_has_comment</i>	Set to 1 if a line has a comment that contains text.
<i>lin_has_code</i>	Set to 1 if a line contains C code.
<i>lin_header</i>	Set to 1 if the current line was obtained from a project header file by means of <code>#include "filename"</code> . Set to 2 if the current line was obtained from a system header file by means of <code>#include &lt;filename&gt;</code> .
<i>lin_include_kind</i>	1 if the line includes a project header, 2 if the line includes a system header.
<i>lin_include_name()</i>	The file name this line includes.
<i>lin_is_comment</i>	Set to 1 if a line has no C code and either contains a comment or is contained within a comment. The comment line must contain text to qualify as a real comment.
<i>lin_is_exec</i>	Set to 1 if a line contains code that is executable.
<i>lin_preprocessor</i>	Set to the 1 if a line is a preprocessor directive ( <i>i.e.</i> begins with #).
<i>lin_suppressed</i>	Set to 1 if compilation of a line has been suppressed by the preprocessor.

<i>lin_is_white</i>	Set to 1 if a line consists entirely of whitespace (tabs & spaces), or is a comment line without any text.
<i>lin_number</i>	Set to the number of the current line, relative to the start of the current file.
<i>lin_source</i>	Set to 1 if the current line was <i>not</i> obtained from a header file.

In addition to these, CodeCheck has predefined variables that compute total lines of code at the end of every statement, function, module, and project.

**Statement variables:**

<i>stm_lines</i>	Set to the number of lines in a statement.
------------------	--

**Function variables:**

<i>Variable</i>	<i>Meaning</i>
<i>fcn_total_lines</i>	Set to the total number of lines in the definition of a C function ( <b>statistic</b> ).
<i>fcn_com_lines</i>	Set to the number of pure comment lines in the definition of a C function ( <b>statistic</b> ).
<i>fcn_white_lines</i>	Set to the number of whitespace lines in the definition of a C function ( <b>statistic</b> ).
<i>fcn_exec_lines</i>	Set to the number of executable lines in the definition of a C function ( <b>statistic</b> ).

**Module variables:**

<i>mod_total_lines</i>	Set to the total number of lines in a module ( <b>statistic</b> ).
<i>mod_com_lines</i>	Set to the number of pure comment lines in a module ( <b>statistic</b> ).
<i>mod_white_lines</i>	Set to the number of whitespace lines in a module ( <b>statistic</b> ).

*mod\_exec\_lines*            Set to the number of executable lines in a module  
(**statistic**).

**Project variables:**

*prj\_total\_lines*            Set to the total number of lines in a module.

*prj\_com\_lines*             Set to the number of pure comment lines in a module.

*prj\_white\_lines*          Set to the number of whitespace lines in a module.

*prj\_exec\_lines*            Set to the number of executable lines in a module.

A CodeCheck rule which reports the size of a module as measured in lines of code, using the definition given at the beginning of this section, can be constructed in this way:

```
1  if ( mod_begin
2      n = 0;
3
4  if ( mod_end )
5      {
6      n = mod_total_lines - mod_com_lines - mod_white_lines;
7      printf( "Lines of code in %s: %d", mod_name(), n );
8      }
```

**6.1.3 Statements**

A natural way to overcome some of the problems inherent in the definition of a line of C code is to count executable statements instead. Statements have several advantages over lines: there is very little ambiguity as what constitutes a C statement, and every C statement is a conceptually complete piece of code. Furthermore, the statement is the smallest unit of C programming for which this can be said.

What little ambiguity there is in the definition of a “statement” concerns two matters — how to count statements that include other statements, and whether (or how) to count such non-executable statement-like entities as type definitions, declarations, initializers, and preprocessor directives. Kernighan & Ritchie (KR88:236) identify five kinds of executable C statements:

- *Expression*            an expression followed by a semicolon.
- *Jump*                    a `break`, `continue`, `return`, or `goto` statement.
- *Compound*             a list of statements surrounded by braces.
- *Selection*             an `if`, `if-else`, or `switch` statement.
- *Iteration*             a `for`, `while`, or `do` statement.
- *Labeled*                any statement with a label.

Thus Kernighan & Ritchie do not consider type definitions, declarations, and preprocessor directives as statements at all, and would count this example

```

1  if ( x > y )
2      x = y;
3  else
4      {
5  skip:
6      x = 0;
7      y++;
8      }

```

as six statements (an `if-else` statement, a compound statement, three expression statements, and one labeled statement).

Unfortunately, the Kernighan & Ritchie statement categories are now somewhat out-of-date because C++ has blurred the formerly clear distinction between declarations and executable statements. For our purposes it seems necessary to include declarations as a kind of statement. A simple and useful alternative set of statement categories is as follows:

- *Non-executable*        Local declarations, *except* local C++ declarations that have initializers (because they are executable).
- *Low-level*             Expression and jump statements, and local C++ declarations with initializers.
- *High-level*            Compound, selection, and iteration statement and try blocks ( C++ exception handling ).

In this scheme an `else`-clause is considered to be a separate high-level statement, and labeled statements are not counted twice. Incidentally, CodeCheck considers a label without an accompanying statement (*e.g.* just before the last brace of a function definition) to be a low-level empty statement, equivalent to a labeled semicolon. Such statements are not allowed by the ANSI standard, but are permitted by virtually every compiler.

To provide sufficient flexibility for the definition of a great variety of size measures based on counting statements, CodeCheck offers the following predefined variables. Each variable is set to its appropriate value when the end of the statement (as defined by Kernighan & Ritchie) has been found.

**Statement variables:**

<i>stm_cases</i>	Set to the number of <code>case</code> labels attached to this statement (includes the <code>default</code> label).
<i>stm_catches</i>	Number of handlers in try block
<i>stm_cp_begin</i>	When the open curly brace of a compound statement has been found, this variable is set to the context of the compound statement (see the Reference Manual, section 3.12 for details).
<i>stm_end</i>	Set to 1 when the end of a statement has been found.
<i>stm_end_tryblock</i>	1 when end of whole try-block is reached .
<i>stm_is_expr</i>	Set to 1 if this is an expression statement.
<i>stm_is_jump</i>	Set to 1 if this is a jump statement.
<i>stm_is_comp</i>	When the close curly brace of a compound statement has been found, this variable is set to the context of the compound statement (see the Reference Manual, section 3.12 for details).
<i>stm_is_select</i>	Set to 1 if this is a selection statement.
<i>stm_is_iter</i>	Set to 1 if this is an iteration statement.
<i>stm_is_low</i>	Set to 1 if this is an expression or jump statement.
<i>stm_is_high</i>	Set to 1 if this is a compound, selection, or iteration statement.
<i>stm_is_nonexec</i>	Set to 1 if this is a local declaration. This does <b>not</b> trigger on a local C++ declaration that has an initializer.
<i>stm_labels</i>	Set to the number of ordinary labels (not including <code>case</code> or <code>default</code> labels) attached to this statement.

In some cases, an exception handler may never be reached because the other handler(s) before it can catch the same exception.

*stm\_never\_caught*      1 if an exception handler will never be reached.

In addition to these, CodeCheck has predefined variables that compute total numbers of statements at the end of every function, module, and project:

**Function variables:**

*fcn\_low*                      Set to the number of low-level statements found in the definition of a C function (**statistic**).

*fcn\_high*                     Set to the number of high-level statements found in the definition of a C function (**statistic**).

*fcn\_nonexec*                Set to the number of non-executable statements found in the definition of a C function (**statistic**).

**Module variables:**

*mod\_low*                      Set to the number of low-level statements found in a module (**statistic**).

*mod\_high*                     Set to the number of high-level statements found in a module (**statistic**).

*mod\_nonexec*                Set to the number of non-executable statements found in a module (**statistic**).

**Project variables:**

*prj\_low*                      Set to the number of low-level statements found in a project.

*prj\_high*                     Set to the number of high-level statements found in a project.

*prj\_nonexec*                Set to the number of non-executable statements found in a project.

A CodeCheck rule which reports the size of a project as measured in statements, using the broadest definition of the term, can be constructed in this way:

```
1  if ( prj_end )
2  {
3    size = prj_low + prj_high + prj_nonexec;
4    printf( "Total project statements = %d", size );
5  }
```

### 6.1.4 Tokens

A third, natural way to overcome some of the problems inherent in measuring program size in terms of lines of code is to count tokens. This is the method pioneered by the late Maurice Halstead under the name “Software Science” (MH77). A token is the smallest unit of text recognized by a compiler, *e.g.* the keyword “**else**”, the operator “**+=**”, and the punctuation mark “**;**”. There are, however, two kinds of tokens in C: preprocessor tokens and lexical tokens. The two are not always the same, especially in older versions of C. The ANSI standard has gone a long way towards bringing the two into closer agreement, but some differences still remain (and will always remain as long as C has a preprocessor). To confuse matters even further, Halstead’s tokens do not exactly coincide with the tokens of the C grammar, nor do all users of Halstead’s metrics agree on how his definitions are best implemented.

Fortunately, studies have shown that program size as measured by token counting is not sensitive to minor variations in the details of the definition of tokens (CDS86:42). It is always important, of course, to be consistent when making comparisons of program size, but the actual definition used seems to make very little difference in the conclusions reached.

Halstead divided all tokens into two somewhat arbitrary classes: operators and operands. CodeCheck interprets this classification in the C and C++ contexts as follows: every token that is an identifier, numeric constant, string literal, character literal, or label is classified as an **operand**. A “**Halstead operator**” is any token that is not an operand. Note that standard C operators, enumerated in Sections 2.4 and 6.6, are slightly different. For example, every punctuation mark is a Halstead operator but not a C operator. For those who need measures based on Halstead operators, CodeCheck provides two versions of each predefined variable for counting operators, one using Halstead’s definition, the other using the

standard C definition. For our purposes the term **standard operator** will refer to the standard C operator.

The tokens counted by CodeCheck in this context are *preprocessor* tokens, *i.e.* the tokens seen by the preprocessor *before* macro expansion takes place. To include tokens generated by macro expansion in all of these counts, specify **-E** on the command line.

CodeCheck provides 16 predefined variables with which line, function, module, and program size can be measured in terms of tokens.

**Line variables:**

<i>lin_tokens</i>	Set to the number of tokens found in a line of code before macro expansion.
<i>lin_operators</i>	Set to the number of standard C operators found in a line of code, before macro expansion.
<i>lin_operands</i>	Set to the number of operands found in a line of code, before macro expansion

**Statement variables:**

<i>stm_operators</i>	Set to the total number of standard operators found in a statement, before macro expansion.
<i>stm_operands</i>	Set to the total number of operands found in a statement, before macro expansion.
<i>stm_tokens</i>	Set to the total number of tokens found in a statement, before macro expansion.

**Function variables:**

<i>fcn_H_operators</i>	Set to the total number of Halstead operators found in a function before macro expansion ( <b>statistic</b> ).
<i>fcn_uH_operators</i>	Set to the number of unique Halstead operators found in a function before macro expansion ( <b>statistic</b> ).

<i>fcn_tokens</i>	Set to the total number of tokens found in a function before macro expansion ( <b>statistic</b> ).
<i>fcn_operators</i>	Set to the total number of standard C operators found in a function before macro expansion ( <b>statistic</b> ).
<i>fcn_operands</i>	Set to the total number of operands found in a function, before macro expansion ( <b>statistic</b> ).
<i>fcn_u_operands</i>	Set to the number of unique operands found in a function, before macro expansion ( <b>statistic</b> ).

**Module variables:**

<i>mod_H_operators</i>	Set to the total number of Halstead operators found in a module before macro expansion ( <b>statistic</b> ).
<i>mod_uH_operators</i>	Set to the number of unique Halstead operators found in a module before macro expansion ( <b>statistic</b> ).
<i>mod_tokens</i>	Set to the total number of tokens found in a module before macro expansion ( <b>statistic</b> ).
<i>mod_operators</i>	Set to the total number of standard C operators found in a module before macro expansion ( <b>statistic</b> ).
<i>mod_operands</i>	Set to the total number of operands found in a module, before macro expansion ( <b>statistic</b> ).
<i>mod_u_operands</i>	Set to the number of unique operands found in a module, before macro expansion ( <b>statistic</b> ).

**Project variables:**

<i>prj_H_operators</i>	Set to the total number of Halstead operators found in a project before macro expansion.
<i>prj_uH_operators</i>	Set to the number of unique Halstead operators found in a project before macro expansion.

<i>prj_tokens</i>	Set to the total number of tokens found in a project before macro expansion.
<i>prj_operators</i>	Set to the total number of standard C operators found in a project before macro expansion.
<i>prj_operands</i>	Set to the total number of operands found in a project, before macro expansion.
<i>prj_u_operands</i>	Set to the number of unique operands found in a project, before macro expansion.

### 6.1.5 Functions

A fourth, natural way to overcome some of the problems inherent in measuring program size in terms of lines of code is to count functions and macros (CDS86:42). Of all the measures presented here, this is the simplest and least ambiguous.

CodeCheck provides two predefined variables for counting functions at the module and project level:

<b>Variable</b>	<b>Meaning</b>
<i>mod_functions</i>	Set to the number of functions defined in a module ( <b>statistic</b> ).
<i>mod_macros</i>	Set to the number of macros defined in a module ( <b>statistic</b> ).
<i>prj_functions</i>	Set to the number of functions defined in a project.
<i>prj_macros</i>	Set to the number of macros defined in a project.

## 6.2 Logical Complexity

Measures of logical complexity are extremely important tools for ensuring program maintainability, on the universally recognized principle that logically complex code is very difficult to understand and maintain. Contrary to popular belief, complicated problems do not need logically complex programs for their solution. Indeed, it can be argued that logical complexity in programs is caused by lack of understanding of the problem on the part of the programmer: the better he or she understands the problem, the simpler and more elegant the program produced to solve it. One can argue that every milestone in the history of computer programming has been a technique or concept that reduces program complexity: symbolic variables, formula translation, formatted I/O, structured control statements, recursive functions, structured data, modular programs, and object-oriented programming.

Oddly enough, most measures of logical complexity are not themselves complex. Popular metrics such as decision count (section 6.2.1) and mean logical depth (section 6.2.3) are easy to calculate, while McCabe's cyclomatic complexity number only looks complicated — it is actually easily calculated from the decision count.

### 6.2.1 Decision Count

The flow of control in a C function proceeds sequentially through its statements until it is interrupted by a goto statement, a function call, or a *binary decision point*, a statement from which control passes to one of two choices. Goto statements and function calls are not binary decision points, because each passes control to exactly one choice. A switch statement is not in itself a binary decision point, but each case statement is. Thus every C statement either leads to exactly one statement or is a binary decision point (the switch statement leads to the first case in the switch).

A simple measure for the logical complexity of a function can be constructed by counting the number of binary decision points in the function. This is called the *decision count* for the function.

CodeCheck provides predefined variables for counting decisions at the function, module, and project levels.

<b>Variable</b>	<b>Meaning</b>
<i>fcn_decisions</i>	Set to the number of binary decision points in a function ( <b>statistic</b> ).
<i>mod_decisions</i>	Set to the number of binary decision points in a module ( <b>statistic</b> ).
<i>prj_decisions</i>	Set to the number of binary decision points in a project.

Here is an example set of CodeCheck rules that alerts the user to functions with more than 12 decisions, and calculates the *decisions per function* rate for the entire project:

```

1 float dpf; /* decisions per function */
2
3 if ( fcn_decisions > 12 )
4     warn( "This function may be too complex." );
5
6 if ( prj_end )
7     {
8     dpf = (1.0*prj_decisions) / prj_functions;
9     printf( "Decisions per function rate = %g\n", dpf );
10    }

```

## 6.2.2 McCabe Cyclomatic Complexity

The structure of an algorithm is often depicted by a directed graph called a flowchart or flowgraph. If the flowgraph is abbreviated so that it describes only test nodes and branches between nodes, then it represents the *logic structure* of the algorithm (CDS86:60). Consider the algorithm for computing the greatest common divisor of two positive integers, for example. The following C code implements this simple algorithm:

```

1 long gcd( long n, long d )
2     {
3     register long temp;
4

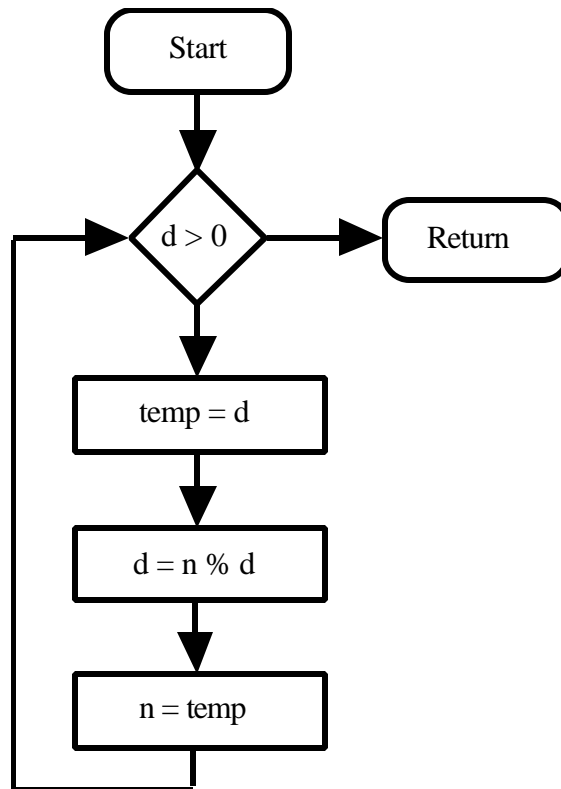
```

```

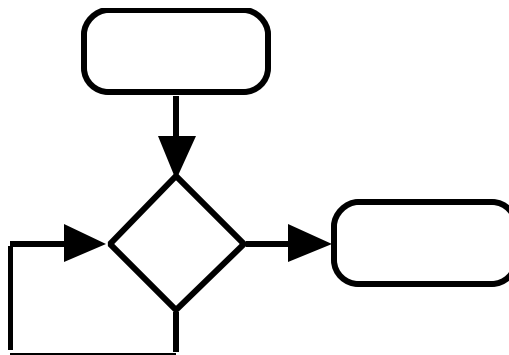
5   while( d )
6     {
7       temp = d;
8       d = n % d;
9       n = temp;
10    }
11    return n;
12   }

```

The flowgraph for this algorithm looks like this:



However, three nodes of this flowgraph are not involved with branching. To obtain its logic structure we collapse these extraneous nodes, which yields a much more simple graph:



By constructing the logic structure graph for this function we have revealed its logical flow in its starkest, most visible form. This is the purpose of the logic structure graph. It is intuitively clear that the logical complexity of a function should be related in some way to the structural richness of this graph. In 1976 McCabe proposed a measure for this structural richness that has been universally accepted.

McCabe's *cyclomatic complexity measure* for a C function or program is derived from simple properties of the logic structure graph of the function. This measure is defined as follows. First connect all exit nodes to the entry node. Then calculate

$$\text{edges} - \text{nodes} + 1$$

where "edges" are lines that connect boxes, and "nodes" are boxes that emit or receive edges.

In the greatest common divisor algorithm, described above, McCabe's measure is easily calculated from the logic structure graph in the above figure as:  $4 \text{ edges} - 3 \text{ nodes} + 1 = 2$ . It is interesting to note that the same result is found if one calculates the McCabe measure on the flowgraph itself:  $7 \text{ edges} - 6 \text{ nodes} + 1 = 2$ . That this is true for any algorithm reflects a fundamental principle: the logical complexity of an algorithm does not change when non-decision nodes are inserted or removed.

The simplest possible algorithm has a McCabe measure of 1, and McCabe considered that cyclomatic complexities in excess of 10 were an indication of overly complex code. One way to understand cyclomatic complexity is this: first imagine that all exit nodes are connected to the entry node. Then count the number of edges that must be removed from the logic structure graph to eliminate all circuits (ignore arrow directions when looking for circuits). The number of edges removed is the cyclomatic complexity.

There is a remarkable relationship between McCabe's cyclomatic complexity and the decision count metric defined in the previous section: *McCabe's number for any algorithm is exactly one more than its decision count*. This occurs because each binary decision point contributes exactly two edges and one node, thus increasing the McCabe measure by one.

Because of the simple relationship between McCabe's measure and decision count, no additional CodeCheck variables are needed to calculate the McCabe measure.

## 6.2.3 Logical Depth

The “logical depth” of a statement is its logical nesting level. Logical nesting results from decisions, *i.e.* `if`, `else`, `for`, `while`, `switch` and `do` statements. The following code fragment, taken from a full-pivot matrix inversion routine, illustrates a nesting depth of 5. Note that there are a total of 12 statements in these 14 lines, consisting of 5 expression statements, 2 iteration statements, 3 selection statements, and 2 compound statements.

```

                                                    /* DEPTH */
1  max = 0;                                       /*    0   */
2  for ( j = 0; j < dim; j++ )                   /*    0   */
3    if ( notused[j] )                           /*    1   */
4      for ( k = 0; k < dim; k++ )               /*    2   */
5        if ( notused[k] )                       /*    3   */
6          {                                     /*    4   */
7            temp = fabs( A[j,k] );               /*    4   */
8            if ( max <= temp )                   /*    4   */
9              {                                  /*    5   */
10             row = j;                           /*    5   */
11             col = k;                           /*    5   */
12             max = temp;                         /*    5   */
13           }                                    /*    5   */
14         }                                     /*    4   */
```

The complexity caused by nesting may be understood by trying to answer this question: when all the looping is done, what are `row`, `col` and `max` set to? Clearly, the greater the depth the harder it is to figure out the answer to such questions.

Functions, which consist of many statements, have several kinds of depth measures. Two measures of obvious utility are *maximum depth* and *average depth*. Since these are statistical quantities, CodeCheck provides a predefined variable from which these and other statistical descriptors can be derived.

### **Variable**

### **Meaning**

`stm_depth`

Set to the logical depth of a statement, *i.e.* its nesting level within *if*-, *for*-, *while*-, and *do*-statements.

Here is an example of a set of CodeCheck rules that calculates a histogram of maximum depth for the functions in each module:

```
1  statistic float   depth, maxdepth;
2  int          max;
3
4  if ( stm_end )           /* Needed because stm_depth */
5     depth = stm_depth;    /* is not a statistic.    */
6
7  if ( fcn_begin )        /* Initialize in */
8     reset( depth );      /* every function */
9
10 if ( fcn_end )
11     maxdepth = maximum( depth );
12
13 if ( mod_begin )        /* Initialize in */
14     reset( maxdepth );  /* every module  */
15
16 if ( mod_end )
17     {
18     max = maximum( maxdepth );
19     histogram( maxdepth, 0, max, max );
20     }
```

## 6.3 Code Density

Like the measures of logical complexity discussed in the previous section, measures of code density are important tools for ensuring program maintainability, on the realistic principle that dense code is very difficult to understand and maintain. Intuitively speaking, computer code is *dense* if it contains a lot of material per unit of size. Size has been discussed in section 6.1 above, but what about “material”? There are many possibilities: number of operators, number of operands, tokens per line, operators per hundred tokens, *etc.*, and these may be weighted according to their cognitive ambiguity or rarity (generally speaking, the more ambiguous or rare an object, *e.g.* a comma operator, the more it should contribute to the density measure). The subsections that follow describe these and other useful ideas for measuring code density.

Code density may be interesting as an overall measurement of a C program, but it can also serve another purpose, which will be even more useful for many programmers. Just as indicators of code complexity can be used to alert the programmer to functions that are overly complex, an indicator of code density can be used to identify lines or statements that are overly dense. Thus a kind of early warning system can be built by writing a CodeCheck rule file that contains a variety of alerts of this type.

### 6.3.1 Two Examples of Dense Code

This C function, adapted from KR88:49, illustrates the value of a warning based on density:

```
1  /* getbits:  get n bits from position p  */
2  unsigned getbits(unsigned x, int p, int n)
3  {
4      return (x>>(p+1-n))&~(~0<<n);
5  }
```

This routine has just one executable statement. How long will it take a maintenance programmer to understand how it works? Five minutes? Ten? Most of the difficulty comes from the uncommented use of so many operators, five of which are rarely-seen bitwise operators. The calculation should be broken down into intermediate steps — this will not only help the reader to parse the expression correctly, it will also provide space in which to place comments. CodeCheck

rules for any of the following three excess-density conditions would have flagged this line:

- a) operators per line > 5,
- b) tokens per line > 15,
- c) operators per operand > 4/3.

Here is a preferable version, which uses register variables to achieve a speed at least as good as the original, if not better (the actual speed will depend on the amount of optimization performed by the compiler):

```
1  #define  RIGHTBITS(n)  (~(0 << n))  /* Mask for right n bits */
2
3  /* getbits: get n bits from position p (counting p from right) */
4
5  unsigned getbits(unsigned x, int p, int n)
6  {
7      register unsigned margin, /* # unwanted bits on right */
8                          shifted, /* result after rt shifting */
9                          mask; /* mask for the desired bits */
10
11     margin = p + 1 - n; /* size of the right-hand margin */
12     shifted = x >> margin; /* shift wanted bits to the right */
13     mask = RIGHTBITS(n); /* create a mask for right n bits */
14     return (shifted & mask); /* mask out all unwanted bits */
15 }
```

Each line of the new version has a lower density by almost any measure, and none would be flagged by the excess-density conditions listed above. In addition, the extra space gained at the end of the new lines allows room for brief comments, which, at least in this example, are crucial.

Here is another example of overly dense and abstract code, also adapted from K&R88. This routine implements the library function *strcpy*:

```
1  /* strcpy: copy string at t to s. */
2
3  void strcpy(char *s, char *t)
4  {
5      while (*s++ = *t++)
6          ;
7  }
```

The apparent elegance of this routine is only superficial — as a programming puzzle it is superb, but as a working routine it exhibits almost every cognitive difficulty that a C routine can present. To fully comprehend how it works, the reader must *unnecessarily* go through a long series of analytical steps, each requiring insight, perceptiveness, and precision. First, the reader must perceive

that the condition has an assignment operator (=), not an equality test (==). Second, the reader must separate the lvalue (\*s) from the increment operator (++), and figure out whether the increment is applied to \*s or to s. Third, the rvalue must be understood: it looks just like the lvalue, but it yields a character, not an address of a character, so it is actually different. Without perceiving this difference no sense can be made of the expression. Fourth (and last), the reader must distinguish between the action taken by the expression and the test for termination of the while statement. The termination test is only understood when the reader makes the identification between logical *false* and the *end-of-string* marker.

Kernighan & Ritchie admit that this routine is indeed cryptic at first sight. However, they suggest that it uses an idiom that should be mastered due to its notational convenience and frequent use (KR88:106). Is convenience a good substitute for clarity? Very few quality control managers would agree.

### 6.3.2 Operational Density

Any metric for code density that uses an operator count in the numerator is a measure of operational density. The rationale for making *operators* the basis for measuring code density is that it is the operators that encode actions: too many actions in too small a space is not good programming style. However, there are many ways to use operator counts in the construction of density metrics. CodeCheck offers a variety of predefined variables with which to calculate operator-based density. First, here are some of the many possible metrics for operational density:

<i>operators per executable line</i>	Used primarily to identify lines that ought to be broken down into several simpler lines.
<i>operators per expression</i>	Used primarily to identify expressions that ought to be simplified.
<i>operators per hundred tokens</i>	Can be used to quantify the overall density of functions, modules, and projects.
<i>operators per operand</i>	Can be used to quantify the “abstractness” of expressions, lines, statements, and functions.

These and other metrics can be constructed with the following CodeCheck predefined variables.

### **Expression variables:**

<b>Variable</b>	<b>Meaning</b>
<i>exp_operators</i>	Set to the number of standard C operators found in an expression, before macro expansion.
<i>exp_operands</i>	Set to the number of operands found in an expression, before macro expansion.
<i>exp_tokens</i>	Set to the number of tokens found in an expression, before macro expansion.

### **Line variables:**

<i>lin_has_code</i>	Set to 1 if a line contains C code.
<i>lin_is_exec</i>	Set to 1 if a line contains code that is executable.
<i>lin_operators</i>	Set to the number of standard C operators found in a line of code, before macro expansion.
<i>lin_operands</i>	Set to the number of operands found in a line of code, before macro expansion.
<i>lin_tokens</i>	Set to the number of tokens found in a line, before macro expansion.

### **Statement variables:**

<i>stm_operators</i>	Set to the number of standard C operators found in a statement, before macro expansion.
<i>stm_operands</i>	Set to the number of operands found in a statement, before macro expansion.
<i>stm_tokens</i>	Set to the number of tokens found in a statement, before macro expansion.

### **Function variables:**

<i>fcn_operators</i>	Set to the number of standard C operators found in a function before macro expansion ( <b>static</b> ).
----------------------	---

*fcn\_operands* Set to the number of operands found in a function, before macro expansion (**statistic**).

*fcn\_tokens* Set to the number of tokens found in a function, before macro expansion (**statistic**).

### **Module variables:**

*mod\_operators* Set to the number of standard C operators found in a module before macro expansion (**statistic**).

*mod\_operands* Set to the number of operands found in a module, before macro expansion (**statistic**).

*mod\_tokens* Set to the number of tokens found in a module before macro expansion (**statistic**).

### **Project variables:**

*prj\_operators* Set to the number of standard C operators found in a project before macro expansion.

*prj\_operands* Set to the number of operands found in a project, before macro expansion.

*prj\_tokens* Set to the number of tokens found in a project before macro expansion.

Here is a sample CodeCheck rule set that calculates five of the operator-based density metrics mentioned above. Note the use in line 18 of multiplication by 1.0 to achieve the effect of casting to float (casts are not permitted in CodeCheck rules).

```
1 float oo_ratio, ops_per_token;
2
3 if ( lin_is_exec )
4     if ( lin_operators > 5 )
5         warn( "This line is too dense." );
6
7 if ( exp_operators > 5 )
8     warn( "This expression is too dense." );
9
10 if ( lin_operands > 2 )
11     if ( 2*lin_operators > 3*lin_operands )
```

```

12     warn( "This line is too abstract." );
13
14     if ( fcn_end )
15     {
16         printf( "Characteristics of %s:\n\n", fcn_name() );
17
18         oo_ratio = (1.0*fcn_operators)/fcn_operands;
19         printf( " Operator/operand ratio = %g\n", oo_ratio );
20
21         ops_per_token = (100.0*fcn_operators)/fcn_tokens;
22         printf( " Operators per hundred tokens = %g\n, ops_per_token );
23     }

```

### 6.3.3 A Weighted Index for Operational Density

Not all operators are equal with respect to frequency of use and familiarity to programmers. Many excellent C programmers can easily spend a career without ever using the comma operator, for example. Furthermore, some operators, like % and >>, are unreliable, due to being somewhat machine-specific. Thus it makes sense to weight operators in an operational density metric according to general familiarity and reliability, on the grounds that the more the reader has to think about an operator and how it behaves, the more it contributes to the overall density of the code.

A simple weighting scheme for operators can be constructed by assigning integer weights to each category of operator. Then every time an operator is found, add its weight to the operator count. The result is a *weighted index* instead of a count. For greater precision, floating-point weights can be used. The crucial step, however, is the initial choice of weights. Here is an example:

Operators	Weight	Justification
+ - * / !	1	These are the best known operators.
[ ] .	1	Very common.
< <= > >=	1	Very common.
!=  =	2	Easily confused.
+= -= *= /=	2	Require a little additional thought.
++ --	3	Need thought, and they resemble + and -.

= ==	3	Notoriously ambiguous.
&&	2	Common, but resemble & and  .
&   ~ <<	3	Not often used.
>> >>= % %=	5	Machine-specific.
* & ->	3	Pointer operations require thought.
sizeof	2	Argument is an unevaluated expression.
( <i>type name</i> )	2	Casts need abstract declarators.
^ ?: ^=	3	Requires thought.
,	5	The comma is very difficult to recognize.

Managers, software engineers, and researchers will differ on how to assign weights to operators. To allow individual weighting choices, CodeCheck provides a separate predefined variable for every distinct operator and punctuation mark (as we have seen before, in the case of the Halstead metrics, one person's punctuation mark may be another person's operator). These variables are set after all macros have been expanded. For the complete list of all operator variables, see the CodeCheck Reference Manual, section 3.9.

Here is a sample CodeCheck rule set that calculates a weighted index for operational density at the module level, using the weights suggested earlier. It defines operational density as *weighted operators per executable line*.

```

1 float index;          // The weighted index */
2
3 if ( mod_begin ) // Initialize at start of each module
4     index = 0;
5
6 if ( op_add || op_subt || op_mul || op_div || op_log_not )
7     index += 1.0;
8
9 if ( op_subscript || op_call || op_member )
10    index += 1.0;
11
12 if ( op_less || op_less_eq || op_more || op_more_eq )
13    index += 1.0
14
15 if ( op_not_eq || op_or_assign )
16    index += 2.0;

```

```
17
18 if ( op_add_assign || op_sub_assign )
19     index += 2.0;
20
21 if ( op_mul_assign || op_div_assign )
22     index += 2.0;
23
24 if ( op_pre_incr || op_pre_decr
25     || op_post_incr
26     || op_post_decr )
27     index += 3.0;
28
29     ... // more index calculations
30
31 if ( op_comma )
32     index += 5.0;
33
34 if ( mod_end ) /* Print at end of each module */
35     {
36     index /= mod_exec_lines;
37     printf( "Module %s:\n", mod_name() );
38     printf( "    operational density = %g\n, index );
39     }
```

# Chapter 7: CodeCheck Rule Sets

## 7.1 Verifying POSIX.1 compliance

The rule file shown here is designed to flag non-POSIX features that are likely to occur in C programs written for Berkeley Unix (BSD version 4.3).

### Rule file *bsd43.cc*

```
/* BSD43.cc

Copyright (c) 1992-95 by Abraxas Software. All rights reserved.
=====
Purpose:  Flags BSD 4.3 features that are not POSIX.1 conforming.
Author:   Loren Cobb.
Revision: 12 October 1994.
          16, March, 1994. Mask the message for the functions
          which are class member functions and have the same
          names in the list.
Format:   Monospaced font with 4 spaces/tab.
=====
```

#### Abstract:

These CodeCheck rules generate warning messages when BSD 4.3 features are used that are not POSIX conforming. If possible, these messages will suggest the appropriate POSIX feature to use.

The features flagged in these rules are the ones that I know about as of the date on this rule file. If you know of others that ought to be flagged, please fax me the details at your earliest convenience. The Abraxas fax number is 503-244-8375. Many thanks in advance!

#### Warning Codes:

```
2000 Precede all headers with #define _POSIX_SOURCE.
2001 Replace function <BSD function> with <POSIX function>.
2002 Function <BSD function> has no POSIX equivalent.
2003 Function <BSD function> is not needed in POSIX.
2004 POSIX requires #include <POSIX header> for <function>.
2005 Replace <BSD header> with <POSIX header>.
2006 Replace <BSD macro> with <POSIX macro>.
2007 If you need an audible alarm, use \a instead of \07.
2008 Replace tag <BSD name> with <POSIX name>.
```

#### Suggested Actions:

- 2000 The macro `_POSIX_SOURCE` must be defined for POSIX headers to be read correctly. Define this macro at the top of every source file.
- 2001 Look up the replacement function in a POSIX reference - the return type and some arguments may differ from the BSD function, and a POSIX header may need to be included.
- 2002 You will need to hand-code a replacement function.
- 2003 No call to this function is needed in a standard POSIX environment.
- 2004 Insert the appropriate `#include` after `#define _POSIX_SOURCE`.
- 2005 Change the name of the BSD header to the appropriate POSIX header.
- 2006 Change the name of the BSD macro to the appropriate POSIX macro.
- 2007 If you need an audible alarm, change `\07` to `\a` in the string.
- 2008 Change the name of the BSD tag to the appropriate POSIX tag.

Useful References:

Horton, Mark R. (1990) "Portable C Software." Published by Prentice-Hall, Englewood Cliffs, NJ 07632, USA.

IEEE (1988) "IEEE Standard Portable Operating System Interface for Computer Environments 1003.1-1988." Published by IEEE, 345 East 47th Street, New York, NY 10017, USA.

Lewine, Donald A. (1991) "POSIX Programmer's Guide." Published by O'Reilly & Associates, 632 Petaluma Avenue, Sebastopol, CA 95472, USA.

Zlotnick, Fred (1991) "The POSIX.1 Standard." Published by Benjamin/Cummings, 390 Bridge Parkway, Redwood City, CA 94065, USA.

```
=====
*/
```

```
#define      NEED_POSIX      2000
#define      REPLACE_FCN     2001
#define      NO_EQUIV       2002
#define      NOT_NEEDED     2003
#define      INCLUDE        2004
#define      REPLACE_HDR    2005
#define      REPLACE_CON    2006
#define      ALARM          2007
#define      REPLACE_TAG    2008

#define REPLACE(fname,gname)  if ( strcmp(idn_name(),fname) == 0 )      \
    {                                                                    \
        \                                                                    \
        warn(REPLACE_FCN, "Replace " fname " with " gname ".");        \
                                                                    \
        ++items_flagged;                                                                    \
    }

#define REWRITE(fname)  if ( strcmp(idn_name(),fname) == 0 )
    \
    {
    \
```

```

        warn(NO_EQUIV, "Function " fname " has no POSIX equivalent."); \
        ++items_flagged;
    \
}

#define DELETE(fname)  if ( strcmp(idn_name(),fname) == 0 )
    \
    {
    \
    warn(NOT_NEEDED, "Function " fname " is not needed in POSIX."); \
    ++items_flagged;
    \
    }

#define CALLED(fname)  (strcmp(idn_name(),fname) == 0)

int    posix_needed,    // 1 if macro _POSIX_SOURCE has not yet been defined.
unistd_needed,        // 1 if header unistd.h has not yet been included.
items_flagged;        // Number of non-POSIX features found in this
module.

int    sys_dir_included, // 1 if header sys/dir.h has been #included.
sys_time_included,     // 1 if header sys/time.h      has been #included.
time_included,         // 1 if header time.h        has been #included.
unistd_included;       // 1 if header unistd.h   has been #included.

if ( mod_begin )
{
    posix_needed      = 1;
    unistd_needed     = 1;
    items_flagged     = 0;
    sys_dir_included  = 0;
    sys_time_included = 0;
    time_included     = 0;
    unistd_included   = 0;
}

if ( pp_macro )
    if ( strcmp(pp_name(), "_POSIX_SOURCE") == 0 )
        posix_needed = 0;

if ( header_name() )
{
    if ( posix_needed )
    {
        warn( NEED_POSIX, "Precede all headers with #define _POSIX_SOURCE" );
        posix_needed = 0;
        ++items_flagged;
    }

    if ( unistd_needed )
    {
        if ( strcmp(header_name(), "unistd.h") != 0 )

```

```

        warn( INCLUDE, "POSIX recommends #include <unistd.h> before this
line." );
        unistd_needed = 0;
        ++items_flagged;
    }

    if ( strcmp(header_name(),"sys/dir.h") == 0 )
    {
        warn( REPLACE_HDR, "Replace <sys/dir.h> with <dirent.h>." );           //
MRH:328
        ++items_flagged;
    }
    else if ( strcmp(header_name(),"sys/param.h") == 0 )
    {
        warn( REPLACE_HDR, "Replace <sys/param.h> with <unistd.h>." );
        ++items_flagged;
    }
    else if ( strcmp(header_name(),"sys/time.h") == 0 )
    {
        warn( REPLACE_HDR, "Replace <sys/time.h> with <time.h>." );
        ++items_flagged;
        sys_time_included = 1;
    }
    else if ( strcmp(header_name(),"unistd.h") == 0 )
    {
        unistd_included = 1;
        unistd_needed = 0;
    }
    else if ( strcmp(header_name(),"varargs.h") == 0 )
    {
        warn( REPLACE_HDR, "Replace <varargs.h> with <stdarg.h>." );
        ++items_flagged;
    }
}

if (idn_function&&!idn_member)
{
    REPLACE( "alloca"           , "malloc"
) // DAL:566
    else REPLACE( "bcmp"         , "strncmp"
) // DAL:566
    else REPLACE( "bcopy"        , "strncmp"
) // DAL:566
    else REPLACE( "cuserid"      , "getlogin or getpwuid"
) // DAL:249
    else REPLACE( "ecvt"         , "sprintf"
) // DAL:566
    else REPLACE( "fcvt"         , "sprintf"
) // DAL:566
    else REPLACE( "flock"        , "fcntl"
) // DAL:566
    else REPLACE( "gcvt"         , "sprintf"
) // DAL:566
    else REPLACE( "getdtablesize", "sysconf"
) // DAL:566

```

```

else REPLACE( "getpw" , "getpwent" )
// DAL:566
else REPLACE( "gettimeofday" , "localtime and time" )
// DAL:566
else REPLACE( "getwd" , "getcwd"
) // DAL:566
else REPLACE( "index" , "strchr"
) // DAL:566
else REPLACE( "initstate" , "srand"
) // DAL:566
else REPLACE( "ioctl" , "[see a POSIX.1 book]" )
// DAL:566
else REPLACE( "killpg" , "kill"
) // DAL:566
else REPLACE( "mknod" , "mkdir or mkfifo" )
// DAL:566
else REPLACE( "mktemp" , "tmpnam"
) // DAL:487
else REPLACE( "pclose" , "close"
) // DAL:566
else REPLACE( "popen", "pipe, fdopen, fork, system, or wait" )
// DAL:566
else REPLACE( "random" , "rand"
) // DAL:566
else REPLACE( "rindex" , "strrchr"
) // DAL:567
else REPLACE( "scandir" , "readdir, malloc, qsort" )
// DAL:567
else REPLACE( "seekdir" , "opendir, readdir" )
// DAL:567
else REPLACE( "setbuffer" , "setvbuf"
) // DAL:567
else REPLACE( "setitimer" , "alarm"
) // DAL:567
else REPLACE( "setlinebuf" , "setvbuf"
) // DAL:567
else REPLACE( "setregid" , "setgid and setegid" )
// DAL:567
else REPLACE( "setreuid" , "setuid and setuegid" )
// DAL:567
else REPLACE( "setstate" , "srand"
) // DAL:567
else REPLACE( "sigblock" , "sigprocmask"
) // DAL:567
else REPLACE( "signal" , "sigaction" )
// DAL:420
else REPLACE( "sigpause" , "sigsuspend"
) // DAL:567
else REPLACE( "sigsetmask" , "sigprocmask"
) // DAL:567
else REPLACE( "sigvec" , "sigpending"
) // DAL:567
else REPLACE( "srandom" , "srand"
) // DAL:567
else REPLACE( "system" , "[see a POSIX.2 book]" )
// DAL:470

```

```

else REPLACE( "timezone"           , "localtime"           )
// DAL:567
else REPLACE( "utimes"             , "utime"               )
) // DAL:567
else REPLACE( "valloc"             , "malloc"              )
) // DAL:567
else REPLACE( "vfork"              , "fork"                )
) // DAL:567
else REPLACE( "vhangup"            , "tcsetattr"          )
// DAL:567
else REPLACE( "wait3"              , "waitpid"             )
) // DAL:567

else REWRITE( "bzero"              ) // DAL:566
else REWRITE( "cabs"               ) // DAL:566
else REWRITE( "ffs"                ) // DAL:566
else REWRITE( "gamma"              ) // DAL:566
else REWRITE( "getpass"            ) // DAL:566
else REWRITE( "hypot"              ) // DAL:566
else REWRITE( "insque"             ) // DAL:566
else REWRITE( "isascii"            ) // DAL:566
else REWRITE( "j0"                 ) // DAL:566
else REWRITE( "j1"                 ) // DAL:566
else REWRITE( "jn"                 ) // DAL:566
else REWRITE( "remque"             ) // DAL:567
else REWRITE( "y0"                 ) // DAL:567
else REWRITE( "y1"                 ) // DAL:567
else REWRITE( "yn"                 ) // DAL:567

else DELETE( "endgrent"            ) // DAL:566
else DELETE( "endpwent"           ) // DAL:566
else DELETE( "nice"                ) // DAL:566
else DELETE( "setgrent"           ) // DAL:567
else DELETE( "setpwent"           ) // DAL:567

if ( CALLED("asctime") && sys_time_included && (! time_included) )
// DAL:217
{
warn( INCLUDE, "POSIX requires #include <time.h> for function asctime."
);
++items_flagged;
}

if ( identifier("direct") ) // MRH:328
if ( sys_dir_included )
{
warn( REPLACE_TAG, "Replace tag \"direct\" with \"dirent\"." );
++items_flagged;
}

if ( macro("L_INCR") ) // DAL:351
{
warn( REPLACE_CON, "Replace L_INCR with SEEK_CUR." );
++items_flagged;
}

```

```

if ( macro("L_SET") ) // DAL:351
{
warn( REPLACE_CON, "Replace L_SET with SEEK_SET." );
++items_flagged;
}

if ( macro("L_XTND") ) // DAL:351
{
warn( REPLACE_CON, "Replace L_XTND with SEEK_END." );
++items_flagged;
}

if ( macro("O_NDELAY") ) // DAL:366
{
warn( REPLACE_CON, "Replace O_NDELAY with O_NONBLOCK." );
++items_flagged;
}

if ( macro("SIGIOT") ) // DAL:211
{
warn( REPLACE_CON, "Replace SIGIOT with SIGABRT." );
++items_flagged;
}

if ( lex_num_escape == 7 ) // DAL:377
{
warn( ALARM, "If you need an audible alarm, use \\a instead of \\07." );
++items_flagged;
}

if ( mod_end )
{
printf( "\n*** %d non-POSIX BSD features were found in module %s ***\n\n",
items_flagged, mod_name() );
}

```

## 7.2 Compliance with Coding Standards

Almost every company that engages in C software development has its own internal standards for programmers. The rule file that follows encodes the actual standards used by one such company. This rule file is used both by individual programmers and by team leaders to monitor compliance with the company's standards.

### **Rule file** *sample.cc*

```
// Copyright (c) 1988-93 by Abraxas Software. All rights reserved.

// This file can be used by company programmers to monitor
// their compliance with corporate standards.

// The warnings are intended to be seen in the context of
// a list file. Use: check myproject.ccp -Rsample -L

// Warning codes: 1000 lexical
//                 2000 keyword
//                 3000 preprocessor
//                 4000 declaration
//                 5000 general

// These rules may serve as a basis for customizing your
// own set of company standards for C code production.

#include <check.cch>

/***** Lexical Rules *****/

if ( lex_nonstandard )
    warn( 1001, "Nonstandard character." );

if ( lex_unsigned || lex_float )
    warn( 1002, "Do not use this suffix." );

if ( (lex_radix == 8) || (lex_radix == 16) )
    warn( 1003, "Do not use octal or hex constants." );

if ( lex_str_length > 509 )
    warn( 1005, "String literal too long for ANSI C." );

if ( lex_str_trigraph )
    warn( 1006, "ANSI C will recode the trigraph in this string." );

if ( lex_nested_comment )
    warn( 1007, "Do not nest comments." );

if ( op_low )
```

```

    {
    if ( ! op_white_before )
        warn( 1008, "Put space before operator %s.", token() );
    if ( ! op_white_after )
        warn( 1008, "Put space before operator %s.", token() );
    }

/***** Keyword rules *****/

if ( keyword("int") )
    if ( ! lex_macro_token )
        warn( 2001, "Use INTEGER, short, or long." );

if ( keyword("register") )
    if ( ! lex_macro_token )
        warn( 2002, "Use the REGISTER macro here." );

if ( keyword("volatile") )
    warn( 2003, "The \'volatile\' keyword is not portable." );

/***** Preprocessor Rules *****/

if ( pp_sub_keyword )
    warn( 3001, "Do not substitute preprocessor keywords." );

if ( pp_stack || pp_benign )
    warn( 3002, "Do not redefine macros without an #undef." );

if ( pp_include & 1 )
    warn( 3003, "#include macros are not allowed." );

if ( lex_not_manifest )
    warn( 3004, "Define this constant in a macro." );

if ( pp_arg_paren )
    warn( 3005, "Surround this argument with parentheses." );

if ( pp_comment )
    warn( 3006, "Surround this comment with space." );

if ( pp_white_before )
    warn( 3007, "The # must be in column 1 for portability." );

if ( pp_trailer )
    warn( 3008, "Place these tokens in a comment." );

if ( pp_if_depth > 8 )
    warn( 3009, "#if nesting > 3 is not be portable." );

if ( pp_include_depth > 8 )
    warn( 3010, "Include depth exceeds 8." );

if ( pp_pragma )
    warn( 3011, "Pragmas are generally not portable." );

```

```

if ( pp_arg_multiple )
    warn( 3012, "Possible undesired side-effects may occur here." );

if ( pp_assign )
    warn( 3013, "The \"=\" sign in this macro may be an error." );

if ( pp_keyword )
    warn( 3014, "Warning: this macro redefines a keyword." );

if ( pp_overload )
    warn( 3015, "This identifier conflicts with a macro function name." );

if ( pp_semicolon )
    warn( 3016, "Macro body ends with a semicolon." );

if ( pp_undef )
    warn( 3017, "Use #undef as seldom as possible." );

if ( pp_unstack )
    warn( 3018, "Undefining multiply-defined macros is not portable." );

/***** Declaration Rules *****/

if ( dcl_no_specifier )
    {
    if ( dcl_function )
        warn( 4001, "Explicit function return type required." );
    else
        warn( 4002, "Explicit type specifier required." );
    }

if ( dcl_bitfield_anon )
    warn( 4003, "All bitfields must have names." );

if ( dcl_union_init )
    warn( 4004, "Do not initialize unions." );

if ( dcl_auto_init )
    warn( 4005, "Do not initialize auto structs or arrays." );

if ( dcl_union_bits )
    warn( 4006, "Do not use bitfields in unions." );

if ( dcl_tag_def )
    if ( lin_source && (dcl_base == STRUCT_TYPE) )
        warn( 4007, "Define structures in header files only." );

if ( dcl_oldstyle )
    warn( 4008, "ALWAYS use prototypes." );

if ( idn_no_prototype )
    warn( 4009, "Function %s needs a prototype.", idn_name() );

if ( dcl_need_3dots )
    warn( 4010, "ANSI C requires 3 dots (...) here." );

```

```
if ( dcl_hidden )
    warn( 4011, "This declaration hides another." );
```

```
/****** General Rules *****/
```

```
if ( macro("offsetof" ) )
    warn( 5001, "Do not use offsets!" );
```

## 7.3 Porting to ANSI C

The differences between the *de facto* H&S standard of 1984-89 and the recently published ANSI standard have been described in a number of recent publications, the best of which are RJ88, HS88, and KR88. The rules given here for portation from H&S to ANSI were developed from all of these sources. These rules are not complete, but they cover most of the major problem areas.

### Rule file *ansi.cc*

```
/*  ansi.cc
   Copyright (c) 1992-94 by Abraxas Software. All rights reserved.
   =====
   Purpose:  Checks for compatibility with ANSI C standards.
   Author:   Loren Cobb.
   Revision: 12 October 1994.
   Format:   Monospaced font with 4 spaces/tab.
   =====
```

#### Abstract:

These CodeCheck rules check for compatibility with the ANSI C Standard. These rules are not comprehensive, but they do check for a great many of the troublesome areas of ANSI compliance.

These rules are applied to all headers that are included in double quotes, e.g. `#include "project.h"`. For proper use of these rules, be sure to include system headers in angle brackets: `#include <ctypes.h>`.

DOS and OS/2 only: do not use the `-K1` or `-K0` options with these rules, as this will prevent CodeCheck from parsing (and detecting) non-ANSI keywords such as `near`, `far`, `huge`, `pascal`, etc.

#### Warning Codes:

```
8001 Rule file ansi.cc should not be run with option -K1 or -K2.
8002 Octal digits 8 and 9 are illegal in ANSI C.
8003 The long float type is illegal in ANSI C.
8004 ANSI C files must end with a newline character.
8005 String literal too long for ANSI C.
8006 ANSI C does not expand macros inside strings.
8007 ANSI C will recode the trigraph in this string.
8008 Too many parameters in this macro.
8009 This comment will not paste tokens in ANSI C.
8010 Header file nesting is too deep.
8011 This preprocessor usage is not allowed in ANSI C.
8012 ANSI C does not permit sizeof in directives.
8013 ANSI C does not parse tokens that follow a preprocessor directive.
8014 Not an ANSI preprocessor directive.
8015 This declaration is not valid.
8016 Only 31 chars are significant.
```

8017 Use double instead of long float.  
8018 ANSI C requires 3 dots (...) here.  
8019 Return type for this function should be specified.  
8020 An explicit type is required in ANSI C declarations.  
8021 Local static function declarations are not allowed in ANSI C.  
8022 Zero-length arrays are illegal in ANSI C.  
8023 This function has no prototype.  
8024 Too many macros for some ANSI compilers.  
8025 <specifier> is not an ANSI type specifier.  
8026 For Apple "extended" type use ANSI "long double".  
8027 Non-ANSI type modifier (pascal, cdecl, near, far, huge,  
interrupt, export, loadds, saveregs, based, or fastcall).  
8028 Non-ANSI placement of const or volatile.  
8029 Non-ANSI function type specifier (inline, virtual, pure, pascal,  
cdecl, interrupt, loadds, saveregs, fastcall, or export).  
8030 Nested comments are not allowed in ANSI C.  
8031 The preceding label is not attached to a statement.  
8032 Array <name> must be declared extern.  
8033 Binary constants are not permitted in ANSI C.  
8034 <identifier>: Prefix <string> is reserved by ANSI.  
8035 ANSI C does not use va\_dcl.  
8036 Macro va\_start has two arguments in ANSI C.  
8037 Replace header <header> with <header>.  
8038 Replace function <name> with <name>.  
8039 Replacement of preprocessor commands is not allowed in ANSI C.  
8040 Empty initializers are not allowed in ANSI C.

```

=====
*/

#include <check.cch>

#define REPLACE_HDR(hdr1,hdr2)  if ( strcmp(header_name(),hdr1) == 0 )  \
    warn( 8037, "Replace header " hdr1 " with " hdr2 ".");

#define REPLACE_FCN(fname,gname)  if ( strcmp(op_function(),fname) == 0 )  \
    warn( 8038, "Replace " fname " with " gname ".");

int      ch,                // Character that follows a prefix.
         k,                // counter for dcl_level(k)
         non_ANSI_mod,     // Non-ANSI type modifier flags
         varargs_included; // True if <varargs.h> was included.

if ( prj_begin )
{
    /*
     *   Flags for non-ANSI type and pointer modifiers:
     */
    non_ANSI_mod = ~(CONST_FLAG + VOLATILE_FLAG);

    /*
     *   Make sure that extended keywords are allowed on DOS machines:
     */
#ifdef __MSDOS__
    if ( option('K') < 3 )

```

```

        {
            warn( 8001, "Rule file ansi.cc should not be run with -K1 or -K2." );
        }
#endif
    }

if ( lex_big_octal )
    warn( 8002, "Octal digits 8 and 9 are illegal in ANSI C." );

if ( lex_long_float )
    warn( 8003, "The long float type is illegal in ANSI C." );

if ( lex_nl_eof )
    warn( 8004, "ANSI C files must end with a newline character." );

if ( lex_str_length > 509 )
    warn( 8005, "String literal too long for ANSI C." );

if ( lex_str_macro )
    warn( 8006, "ANSI C does not expand macros inside strings." );

if ( lex_str_trigraph )
    warn( 8007, "ANSI C will recode the trigraph in this string." );

if ( pp_arg_count > 31 )
    warn( 8008, "Too many parameters in this macro." );

if ( pp_comment )
    warn( 8009, "This comment will not paste tokens in ANSI C." );

if ( pp_if_depth > 8 )
    warn( 8010, "Header file nesting is too deep." );

if ( pp_not_ansi )
    warn( 8011, "This preprocessor usage is not allowed in ANSI C." );

if ( pp_sizeof )
    warn( 8012, "ANSI C does not permit sizeof in directives." );

if ( pp_trailer )
    warn( 8013, "ANSI C does not parse tokens that follow a preprocessor
directive." );

if ( pp_unknown )
    warn( 8014, "Not an ANSI preprocessor directive." );

if ( dcl_empty )
    if ( ! dcl_tag_def )
        warn( 8015, "This declaration is not valid." );

if ( dcl_ident_length > 31 )
    warn( 8016, "Only 31 chars are significant." );

if ( dcl_long_float )
    warn( 8017, "Use double instead of long float." );

```

```

if ( dcl_need_3dots )
    warn( 8018, "ANSI C requires 3 dots (...) here." );

if ( dcl_no_specifier )
    {
    if ( dcl_function )
        warn( 8019, "Return type for function %s should be specified.",
dcl_name() );
    else
        warn( 8020, "An explicit type for %s is required in ANSI C.", dcl_name()
);
    }

if ( dcl_static )
    if ( dcl_local && dcl_function )
        warn( 8021, "Local static function declarations are not allowed in ANSI
C." );

if ( dcl_zero_array )
    warn( 8022, "Zero-length arrays are illegal in ANSI C." );

if ( idn_no_prototype )
    warn( 8023, "Function %s has no prototype.", idn_name() );

if ( mod_macros > 1024 )
    warn( 8024, "Too many macros for some ANSI compilers." );

if ( dcl_base == EXTRA_INT_TYPE )
    if ( lin_header != SYS_HEADER )
        warn( 8025, "non-ANSI integer type." );

if ( dcl_base == EXTRA_UINT_TYPE )
    if ( lin_header != SYS_HEADER )
        warn( 8025, "non-ANSI unsigned type." );

if ( dcl_base == EXTRA_FLOAT_TYPE )
    if ( lin_header != SYS_HEADER )
        warn( 8025, "non-ANSI float type." );

if ( dcl_base == EXTRA_PTR_TYPE )
    if ( lin_header != SYS_HEADER )
        warn( 8025, "non-ANSI pointer type." );

if ( dcl_storage_flags & GLOBAL_SC )
    if ( lin_header != SYS_HEADER )
        warn( 8025, "VAX globaldef and globalref are not ANSI type specifiers."
);

if ( keyword("comp") )
    if ( lin_header != SYS_HEADER )
        warn( 8025, "Apple comp is not an ANSI type specifier." );

if ( keyword("extended") )
    if ( lin_header != SYS_HEADER )
        warn( 8026, "For Apple \"extended\" type use ANSI \"long double\"." );

```

```

if ( dcl_variable || dcl_function )
    if ( lin_header != SYS_HEADER )
        {
            k = 0;
            while ( k <= dcl_levels )
                if ( dcl_level_flags(k++) & non_ANSI_mod )
                    warn( 8027, "Non-ANSI type modifier." );
        }

if ( dcl_cv_modifier )
    if ( lin_header != SYS_HEADER )
        warn( 8028, "Non-ANSI placement of const or volatile." );

if ( dcl_function_flags )
    warn( 8029, "Non-ANSI function type specifier." );

if ( lex_nested_comment )
    {
        warn( 8030, "Nested comments are not allowed in ANSI C." );
    }

// If you want CodeCheck to assume that nested comments are okay as
// soon as it finds the first such comment, then enable these 2 lines:

/*
    if ( option('N') == 0 )           // Assume that the rest of this file
        set_option( 'N', 1 );       // will contain nested comments too.
*/
    }

// Although many modern compilers allow labels that are not attached
// to any statement, e.g. at the end of a block, this is not allowed
// by the ANSI standard.

if ( stm_bad_label )
    warn( 8031, "The preceding label is not attached to a statement." );

// Detects local arrays that have no explicit dimension. Some
// pre-ANSI compilers consider such arrays to be implicitly
// external (i.e. the identifier has file scope and external
// linkage). This interpretation is not allowed in ANSI C.

if ( dcl_level(0) == ARRAY )
    if ( dcl_local && (dcl_array_size == -1) && (! dcl_parameter) )
        if ( (! dcl_extern) && (! dcl_initializer) )
            warn( 8032, "Array %s must be declared extern.", dcl_name() );

// Detect Zortech binary constants (e.g. 0b10101001):

if ( lex_radix == 2 )
    if ( lin_header != SYS_HEADER )
        warn( 8033, "Binary constants are not permitted in ANSI C." );

```

```

// Check each external identifier for reserved prefixes:

if ( (dcl_global && ! dcl_static) || pp_macro )
    if ( lin_header != SYS_HEADER )
        {
        if ( prefix("E") )
            {
            ch = root()[0];
            if ( isdigit(ch) || isupper(ch) )
                warn( 8034, "%s: prefix E is reserved by ANSI.", dcl_name() );
            }
        if ( prefix("is") )
            {
            ch = root()[0];
            if ( islower(ch) )
                warn( 8034, "%s: prefix \"is\" is reserved by ANSI.", dcl_name()
);
            }
        else if ( prefix("to") )
            {
            ch = root()[0];
            if ( islower(ch) )
                warn( 8034, "%s: prefix to is reserved by ANSI.", dcl_name() );
            }
        else if ( prefix("LC_") )
            {
            ch = root()[0];
            if ( isupper(ch) )
                warn( 8034, "%s: prefix LC_ is reserved by ANSI.", dcl_name() );
            }
        else if ( prefix("SIG") )
            {
            ch = root()[0];
            if ( isupper(ch) || (ch == '_' ) )
                warn( 8034, "%s: prefix SIG is reserved by ANSI.", dcl_name() );
            }
        else if ( prefix("mem") )
            {
            ch = root()[0];
            if ( islower(ch) )
                warn( 8034, "%s: prefix mem is reserved by ANSI.", dcl_name() );
            }
        else if ( prefix("str") )
            {
            ch = root()[0];
            if ( islower(ch) )
                warn( 8034, "%s: prefix str is reserved by ANSI.", dcl_name() );
            }
        else if ( prefix("wcs") )
            {
            ch = root()[0];
            if ( islower(ch) )
                warn( 8034, "%s: prefix wcs is reserved by ANSI.", dcl_name() );
            }
        }
}

```

```

if ( macro("va_dcl" ) )
    warn( 8035, "ANSI C does not use va_dcl." );

if ( macro("va_start" ) )
    if ( varargs_included )
        warn( 8036, "Macro va_start has two arguments in ANSI C." );

if ( header_name() )
    {
    if ( strcmp(header_name(),"varargs.h") == 0 )
        {
        warn( 8037, "Replace <varargs.h> with <stdarg.h>." );
        varargs_included = 1;
        }
    else REPLACE_HDR( "memory.h",      "string.h" )
    else REPLACE_HDR( "sys/times.h",   "time.h" )
    }

if ( op_call )
    {
    REPLACE_FCN( "cfree",      "free" )
    else REPLACE_FCN( "bcmp",   "strcmp" )
    else REPLACE_FCN( "bzero",  "memset" )
    else REPLACE_FCN( "strpos",  "strchr" )
    else REPLACE_FCN( "strrpos", "strchr" )
    else REPLACE_FCN( "mktemp",  "tmpnam" )
    }

if ( pp_sub_keyword )
    warn( 8039, "Replacement of preprocessor commands is not allowed in ANSI C."
);

if ( exp_empty_initializer )
    warn( 8040, "Empty initializers are not allowed in ANSI C." );

```

## 7.4 Porting to Strict K&R Compilers

The differences between the *de facto* H&S standard of 1984-89 and the original Kernighan & Ritchie standard of 1978 were described by Harbison & Steele in HS84. The rule set given here for porting from ANSI C to K&R C was developed primarily from this source. These rules are not complete, but they cover most of the major problem areas.

### Rule file `toKR.c`

```
/* Copyright (c) 1989-92 by Abraxas Software.
 *
 * This is a collection of CodeCheck rules for
 * testing for portability to strict K&R C compilers.
 */

#include <check.cch>

if ( identifier("entry") )
    warn( 7001, "\"entry\" is a reserved keyword in K&R." );

if ( lex_not_KR_escape )
    warn( 7002, "This escape sequence is not defined in K&R." );

if ( lex_backslash )
    warn( 7003, "This line continuation is not allowed in K&R." );

if ( lex_float || lex_unsigned || lex_long_float )
    warn( 7004, "This suffix is not allowed in K&R." );

if ( (lex_radix == 16) || lex_hex_escape )
    warn( 7005, "Hexadecimal numbers are not defined in K&R." );

if ( lex_str_macro )
    warn( 7006, "K&R compilers do not recognize macros in strings." );

if ( lex_str_concat )
    warn( 7007, "Implicit string concatenation not defined in K&R." );

if ( lex_trigraph )
    warn( 7008, "Trigraphs are not defined in K&R." );

if ( lex_wide )
    warn( 7009, "Wide chars and strings are not defined in K&R." );

if ( pp_error )
    warn( 7010, "The #error directive is not defined in K&R." );

if ( pp_paste )
```

```

    warn( 7011, "The # (paste) operator is not defined in K&R." );

if ( pp_pragma )
    warn( 7012, "The #pragma directive is not defined in K&R." );

if ( pp_stringize )
    warn( 7013, "The ## preprocessor operator not defined in K&R." );

if ( pp_unknown )
    warn( 7014, "This preprocessor directive is not defined in K&R." );

if ( pp_defined )
    warn( 7015, "The \"defined\" function is not defined in K&R." );

if ( pp_elif )
    warn( 7016, "The #elif directive is not defined in K&R." );

if ( keyword("const") )
    warn( 7017, "The \"const\" type is not defined in K&R." );

if ( keyword("volatile") )
    warn( 7018, "The \"volatile\" type is not defined in K&R." );

if ( keyword("signed") )
    warn( 7019, "The \"signed\" type is not defined in K&R." );

if ( dcl_function )
    if ( ! dcl_oldstyle )
        warn( 7020, "Prototypes are not allowed in K&R." );

if ( dcl_3dots )
    warn( 7021, "The 3-dot notation is not allowed in K&R." );

if ( dcl_need_3dots )
    warn( 7022, "Comma after argument list is not allowed in K&R." );

if ( keyword("enum") )
    warn( 7023, "The enumerated type is not permitted in K&R." );

if ( keyword("void") )
    warn( 7024, "The void type is not permitted in K&R." );

if ( dcl_union_init )
    warn( 7025, "Unions cannot be initialized in K&R." );

if ( lin_nested_comment )
    warn( 7026, "Nested comments are not permitted in K&R." );

if ( lex_cpp_comment )
    warn( 7027, "The // comment is not permitted in K&R." );

if ( dcl_extern_ambig )
    warn( 7028, "%s matches another name on 6 chars.", dcl_name() );

```

## 7.5 Measuring Code Complexity

The McCabe Cyclomatic Complexity measure, described in Section 6.2.2, is one of the most commonly used measures of code complexity. Here is a sample CodeCheck rule file that calculates the McCabe and other measures of complexity for every function in a project.

### Rule file *complex.cc*

```
/*
 * Copyright (c) 1989-90 by Abraxas Software.
 *
 * This is the start of a collection of CodeCheck rules
 * for measuring function complexity and operator density.
 *
 * NOTE: A CodeCheck bug (corrected in version 2.07) caused
 * the McCabe measure calculated by these rules to be under-
 * estimated: certain decision points were not counted.
 */

statistic int McCabe;
statistic float density;

if ( prj_begin )
{
    printf( "\n===== %s =====\n", prj_name() );
    printf( "\nDate: %s.\n\n", time_stamp() );
    printf( "Complexity => McCabe's Cyclomatic Complexity.\n" );
    printf( "Density => Operators per executable line.\n" );
    printf( "Asterisks => Function is too complex.\n" );
}

if ( mod_begin )
{
    printf( "\n\n----- %s -----\n\n", mod_name() );
    printf( "FUNCTION Complexity Density\n" );
    reset( fcn_decisions );
    reset( fcn_operators );
    reset( fcn_exec_lines );
    reset( McCabe );
    reset( density );
}

if ( fcn_end )
{
    McCabe = 1 + fcn_decisions;
    printf( "%-16s %3d ", fcn_name(), McCabe );
    if ( McCabe >= 30 )
        printf( "**** " );
}
```

```

else if ( McCabe >= 20 )
    printf( "*** " );
else if ( McCabe >= 10 )
    printf( "* " );
else
    printf( " " );

if ( fcn_exec_lines > 0 )
    density = (1.0*fcn_operators) / fcn_exec_lines;
else
    density = 0.0;
printf( "%9.1f\n", density );
}

if ( mod_end )
    if ( ncases(fcn_exec_lines) > 0 )
        {
        printf("\nFunction Density (operators per executable line):\n");
        printf( " Mean:      %6.2f\n", mean(density) );
        printf( " Std.Dev: %6.2f\n", stdev(density) );

        printf( "\nFunction Complexity (McCabe):\n" );
        printf( " Mean:      %6.2f\n", mean(McCabe) );
        printf( " Std.Dev: %6.2f\n", stdev(McCabe) );
        printf( " Maximum: %6.2f\n", maximum(McCabe) );
        printf( " Histogram:\n" );
        histogram( McCabe, 0, 20, 21 );
        printf( "\n" );
        }

if ( prj_end )
    printf( "\n\n===== END =====\n\n" );

```

## 7.6 Verifying the Order of Module Elements

Many software companies have as part of their coding standards a specific ordering for the elements of every source file, *e.g.* first external declarations, then macros, type definitions, static declarations, and finally, function definitions. It is not difficult to build a rule file that checks for violations of these orderings. The following rules illustrate one method in use at a major software house:

### **Rule file** *order.cc*

```
// These CodeCheck rules illustrate one way of enforcing
// a standard order upon the elements of a C source file.

// Copyright (c) 1991 by Abraxas Software. All rights reserved.
// Author: Loren Cobb, Abraxas Software, February 1991.

// This file detects violations of the following order:

// 1. Header file #include directives.
// 2. External variable declarations.
// 3. Function declarations.
// 4. Macro definitions.
// 5. Type definitions.
// 6. Public variable definitions (external linkage).
// 7. Private variable definitions (static identifiers).
// 8. Function definitions.

////////////////////////////////////
// Module Initialization: //
////////////////////////////////////

int stage;          // Module format stage (range: 0-8)

if ( mod_begin )
    stage = 0;      // Reinitialize stage at start of every module.

////////////////////////////////////
// Module Format Rules: //
////////////////////////////////////

if ( pp_include )
{
    if ( (stage > 1) && (stage != 4) )
        warn( 1001, "This #include of file %s is out of sequence.", header_name() );
    stage = 1;
}
```

```

if ( dcl_global )
  if ( lin_source )
    {
    if ( dcl_function )
      {
      if ( dcl_definition )
        stage = 8;
      else
        {
        if ( stage > 3 )
          warn( 1003, "Function %s is out of sequence.", dcl_name() );
        stage = 3;
        }
      }
    else if ( dcl_extern )
      {
      if ( stage > 2 )
        warn( 1002, "Declaration of %s is out of sequence.", dcl_name() );
        stage = 2;
      }
    else if ( dcl_static )
      {
      if ( stage > 7 )
        warn( 1007, "Variable %s is out of sequence.", dcl_name() );
        stage = 7;
      }
    else
      {
      if ( stage > 6 )
        warn( 1006, "Variable %s is out of sequence.", dcl_name() );
        stage = 6;
      }
    }
}

if ( pp_macro )
  if ( lin_source )
    {
    if ( stage > 4 )
      warn( 1004, "This macro definition is out of sequence." );
      stage = 4;
    }
}

if ( dcl_typedef )
  if ( lin_source )
    {
    if ( stage > 5 )
      warn( 1005, "Typename %s is out of sequence.", dcl_name() );
      stage = 5;
    }
}

```

## 7.7 C++ Rules

There have been several attempts to codify good C++ programming practices in the form of standards for C++ source code, but the definitive work in this area is yet to be written. In the interim, here are a few rules for C++ code that can serve as a basis for further development:

### Rule file *cplusplus.cc*

```
// Copyright (c) 1992-93 by Abraxas Software.

// This is a collection of CodeCheck rules
// for elementary checking of C++ source code.

#include <check.cch>

if ( mod_begin )
    if ( (option('K') < 4) || (option('K') > 7) )
        {
            warn( 1000, "Use C++ for this file! (option -K4)." );
        }

////////////////////////////////////

// 1. Declare a virtual destructor in every base class that
// has a virtual function. (Stroustrup & Ellis, p. 278)

int    has_virtual_function,
       has_virtual_destructor;

if ( tag_begin )
    {
        if ( tag_kind > 1 && ! tag_nested && ! tag_local )
            {
                has_virtual_function = 0;
                has_virtual_destructor = 0;
            }
    }

if ( dcl_virtual )
    {
        if ( dcl_base == DESTRUCTOR_TYPE )
            has_virtual_destructor = 1;
        else if ( dcl_base != CONSTRUCTOR_TYPE )
            has_virtual_function = 1;
    }

if ( tag_end )
    {
        if ( tag_kind > 1 && ! tag_nested && ! tag_local )
            {
```

```

        if ( has_virtual_function && ! has_virtual_destructor )
            warn( 2007, "This class needs a virtual destructor." );
        has_virtual_function = 0;
        has_virtual_destructor = 0;
    }
}

/////////////////////////////////////////////////////////////////
// 2. Use const variables or enumerated constants
//     instead of preprocessor (#define) constants.

if ( pp_const )
    warn( 1001, "Do not use the preprocessor to define constants." );

/////////////////////////////////////////////////////////////////
// 3. A nested tag name should have an explicit scope.

if ( lex_invisible )
    warn( 1002, "Nested tag name %s should be scoped.", token() );

/////////////////////////////////////////////////////////////////
// 4. In C++ the const specifier implies internal linkage,
//     so either specify extern or initialize the constant.

if ( dcl_simple )
    if ( dcl_level_flags(0) & CONST_FLAG )
        if ( ! dcl_parameter && ! dcl_initializer && ! dcl_extern )
            warn( 2001, "Initializer or extern specifier needed." );

if ( dcl_level(0) == POINTER )
    if ( dcl_level_flags(0) & CONST_FLAG )
        if ( ! dcl_parameter && ! dcl_initializer && ! dcl_extern )
            warn( 2001, "Initializer or extern specifier needed." );

/////////////////////////////////////////////////////////////////
// 5. C++ variables should ALWAYS have an explicit type.

if ( dcl_no_specifier )
    {
        if ( dcl_function )
            warn( 2002, "%s: Explicit return type recommended.", dcl_name() );
        else
            warn( 2002, "%s: Explicit type specifier needed.", dcl_name() );
    }

/////////////////////////////////////////////////////////////////
// 6. Default function parameters are much safer than
//     variable argument lists, so use them!

if ( dcl_3dots )
    warn( 2003, "Use default parameters, not variable argument lists." );

/////////////////////////////////////////////////////////////////
// 7. Avoid gratuitous overloading of names in C++.

if ( dcl_hidden )

```

```
warn( 2004, "Identifier %s is already in use.", dcl_name() );
```

## 7.8 Advanced C++ Rules

This rule represents sample in-house quality control standard for C++ that can be used by a typical software team.

### **Rule File:** xyz.cc

```
/*  xyzrule.cc //  Copyright (c) 1992-96 by Abraxas Software.

=====
      Purpose:  Checks for compatibility with XYZ C++ standards.
=====

Abstract:

      These CodeCheck rules check for compatibility with the XYZ C++
      Coding Rules.

      These rules are applied to all headers that are included in double
      quotes, e.g. #include "project.h". For proper use of these rules, be
      sure to include system headers in angle brackets: #include <ctypes.h>.

Warning Codes:

9111 Header filename <name> is not in DOS format.
9121 Class names must begin with XYZxxx_
9131 Private member name <name> must not begin in uppercase.
9131 Private member name <name> must end with an underscore.
9211 Definition of class <name> belongs in a header file.
9212 File <name> needs a leading comment block.
9213 Header file <name> should be wrapped in an #ifndef.
9213 <name> is not the correct wrapper name for this file.
9214 Definition of function <name> must NOT be in a header file.
9221 Public section must come first in class <name>.
9222 Data member <name> of class <name> must be private.
9231 Function <name> is too long to be inlined.
9232 Do NOT use inline within a class definition.
9251 Class <name> needs a default constructor.
9251 Class <name> needs a copy constructor.
9252 Class <name> needs an operator=().
9253 Class <name> has too many constructors (limit is 3).
9254 Class <name> needs a destructor.
9255 Destructor for class <name> must be virtual.
9261 Operator <name> should not be a friend.
9261 Do NOT declare friend functions.
9261 Do NOT declare friend classes.
9271 Do NOT use virtual base classes.
9281 Declare <name> using a typedef name, not a basic C type.
```

9282 Define typedef name <name> in a base class.  
 9293 Define enum <name> in a base class.  
 9311 Declare parameter <name> to be a reference to <type>.  
 9312 Operator <name> should not return an object.  
 9314 Function <name> should not return an object.  
 9315 Reference parameters must come first.  
 9316 Constant member functions should be avoided.  
 9317 Parameter <name> should not be const.  
 9411 Use // comments, not oldstyle C comments.  
 9421 Declare <name> as a const, not a macro.  
 9422 This enumeration needs an enum type name.  
 9431 Global constant <name> should be a class member.  
 9441 Function <name> needs an explicit return type.

=====

```

*/

#include <check.cch>

#define DOT      ('.')      // period character
#define SLASH    ('/')      // forward slash character
#define BACKSLASH ('\\')    // backslash character
#define COLON    (':')      // colon character
#define TILDE    (~)        // tilde character
#define UNDERSCORE ('_')    // underscore character

#define PUBLIC    0
#define PROTECTED 1
#define PRIVATE   2

int      ch, j, k, okay, level,
         is_constant, is_object,
         lin_if_depth, // Holds latest value of pp_if_depth.
         comment_needed, // True until a header file's comment-block is
found.
         define_needed, // True until a header file's wrapper macro is
#define'd.
         public_needed, // True when a class definition begins.
         one_liner_needed, // True when a function is explicitly inlined.
         no_wrap_message, // True if a wrapper-needed message has NOT been given.
         class_has_virtual_function,
         destructor_is_virtual,
         non_ref_parm_found, // True if a non-reference fcn parameter has been
found.
         detect_virtual_base,
         bad_name, // True if the wrapper name is wrong.
         length, ch1, ch2;

// ----- Rule 1.1.1 -----

if ( header_name() ) // A header is about to be opened
{
    if ( pp_include < 3 ) // Header filename is in double quotes

```

```

{
j = 0;
k = 0;
ch = header_name()[k++];
while ( ch != 0 )
{
ch = header_name()[k++];
if ( ch == DOT )
{
break;    // beginning of extension found
}
else
{
j = k;    // count characters before dot
}
if ( ch == BACKSLASH || ch == COLON )
{
j = 0;    // DOS directory or disk marker
}
else if ( ch == SLASH || ch == TILDE )
{
j = 0;    // Unix directory or disk marker
}
}
if ( j > 8 )
{
warn( 9111, "Header filename %s is not in DOS 8.3 format.",
      header_name() );
}
else if ( ch == DOT )
{
j = 0;
ch = header_name()[k+j];
while ( ch != 0 )
{
j++;
ch = header_name()[k+j]; // count chars after dot.
}
if ( j > 3 )
{
warn( 9111, "Header filename %s is not in DOS 8.3 format.",
      header_name() );
}
}
}
}

```

```
// ----- Rule 1.2.1 -----
```

```

if ( tag_begin )
{
// Apply this rule to global classes only:

if ( tag_global && (tag_kind == CLASS_TAG) )

```

```

    {
    if ( ! prefix("XYZ") )
        warn( 9121, "Class names must begin with \"XYZ_\" );
    else if ( strstr(tag_name(), "_") == 0 )
        warn( 9121, "Use an underscore after the class name prefix." );
    }
}

// ----- Rule 1.3.1 -----

if ( dcl_member == 3 )
{
    if ( dcl_access == PRIVATE )
    {
        if ( dcl_variable )
        {
            if ( isupper(dcl_name()[0]) )
                warn( 9131, "Private data member name %s must not begin in
uppercase.",
                    dcl_name() );
            if ( ! suffix("_") )
                warn( 9131, "Private data member name %s must end with an
underscore.",
                    dcl_name() );
        }
        else if ( dcl_function )
        {
            if ( isupper(dcl_name()[0]) )
                warn( 9131, "Private function name %s must not begin in
uppercase.",
                    dcl_name() );
            if ( ! suffix("_") )
                warn( 9131, "Private function name %s must end with an
underscore.",
                    dcl_name() );
        }
    }
}

// ----- Rule 2.1.1 -----

if ( tag_begin )
{
    if ( tag_global && (tag_kind == CLASS_TAG) )
    {
        if ( lin_source )
            warn( 9211, "Definition of class %s belongs in a header file.",
                tag_name() );
    }
}
}

```

```

// ----- Rules 2.1.2 -----

if ( mod_begin )
{
    comment_needed = FALSE;
}

if ( lin_end )
{
    if ( lin_number == 1 )
    {
        if ( lin_is_comment )
        {
            comment_needed = FALSE;
        }
        else if ( lin_header )
        {
            comment_needed = TRUE;
        }
    }

    if ( comment_needed )
    {
        if ( lin_is_comment )
        {
            comment_needed = FALSE;
        }
        else if ( lin_preprocessor )
        {
            warn( 9212, "File %s needs a leading comment block.",
                file_name() );
            comment_needed = FALSE;
        }
        else if ( lin_has_code )
        {
            warn( 9212, "File %s needs a leading comment block.",
                file_name() );
            comment_needed = FALSE;
        }
    }
}

```

```

// ----- Rule 2.1.3 -----

```

```

if ( pp_if_depth || pp_endif )
{
    lin_if_depth = pp_if_depth;
}

if ( pp_include )
{
    if ( lin_header )
    {

```

```

        if ( no_wrap_message )
        {
            warn( 9213, "Header file %s should be wrapped in an #ifndef.",
                file_name() );
        }
    }

    no_wrap_message = TRUE;
    define_needed = TRUE;
}

if ( lin_dcl_count )
{
    if ( lin_header && lin_if_depth == 0 )
    {
        if ( no_wrap_message )
        {
            warn( 9213, "Header file %s should be wrapped in an #ifndef.",
                file_name() );
            no_wrap_message = FALSE;
            define_needed = FALSE;
        }
    }
}

if ( pp_macro )
{
    if ( lin_header )
    {
        if ( no_wrap_message && lin_if_depth == 0 )
        {
            warn( 9213, "Header file %s should be wrapped in an #ifndef.",
                file_name() );
            no_wrap_message = FALSE;
            define_needed = FALSE;
        }

        if ( define_needed && lin_if_depth == 1 )
        {
            bad_name = FALSE;
            length = strlen( pp_name() );
            k = 0;
            while ( k < length )
            {
                ch1 = pp_name()[k];
                ch2 = file_name()[k];
                k++;
                if ( isalpha(ch2) )
                {
                    ch2 = toupper( ch2 );
                }
                else if ( ch2 == DOT )
                {
                    ch2 = UNDERSCORE;
                }
                if ( ch1 != ch2 )
                {

```

```

        warn( 9213, "%s is not the correct wrapper name for this
file.",
            pp_name() );
        break;
    }
}
define_needed = FALSE;
}
}

// ----- Rule 2.1.4 -----

if ( dcl_function )
{
    if ( dcl_member && dcl_definition )
    {
        if ( ! lin_source )
        {
            warn( 9214, "Definition of function %s must NOT be in a header file.",
                dcl_name() );
        }

        one_liner_needed = dcl_inline;        //    for Rule 2.3.2
    }
}

// ----- Rule 2.2.1 -----

if ( tag_begin )
{
    if ( tag_global && (tag_kind == CLASS_TAG) )
    {
        public_needed = TRUE;
    }
}

if ( dcl_member )
{
    if ( dcl_access )        //    true if protected or private member
    {
        if ( public_needed )
        {
            warn( 9221, "Public section must come first in class %s.",
                class_name() );
        }
    }
    public_needed = FALSE;
}
}

```

```

// ----- Rule 2.2.2 -----

if ( dcl_member )
{
    if ( dcl_variable && (dcl_access != PRIVATE) )
    {
        warn( 9222, "Data member %s of class %s must be private.",
            dcl_name(), class_name() );
    }
}

// ----- Rule 2.3.1 -----

if ( fcn_exec_lines > 1 )
{
    if ( one_liner_needed )
    {
        warn( 9231, "Function %s is too long to be inlined.",
            fcn_name() );
    }
}

// ----- Rule 2.3.2 -----

if ( dcl_inline )
{
    if ( lin_within_class == 1 )
    {
        warn( 9232, "Do NOT use inline within a class definition. (%d)",
lin_within_class );
    }
}

// ----- Rule 2.4.1 -----

//   *** This needs a new trigger in CodeCheck: dcl_override. ***

// ----- Rules 2.5.1, 2.5.2, and 2.5.4 -----

if ( tag_end )
{
    if ( tag_global && (tag_kind == CLASS_TAG) )
    {
        if ( ! tag_has_default )
        {
            warn( 9251, "Class %s needs a default constructor.", class_name() );
        }
    }
}

```

```

    if ( ! tag_has_copy )
    {
        warn( 9251, "Class %s needs a copy constructor.", class_name() );
    }
    if ( ! tag_has_assign )
    {
        warn( 9252, "Class %s needs an operator=(%s&).", class_name(),
            class_name() );
    }
}

// ----- Rule 2.5.3 -----

if ( tag_constructors > 3 )
{
    warn( 9253, "Class %s has too many constructors (limit = 3).",
        class_name() );
}

// ----- Rules 2.5.4 and 2.5.5 -----

if ( tag_begin )
{
    if ( tag_global && (tag_kind == CLASS_TAG) )
    {
        class_has_virtual_function = FALSE;
        destructor_is_virtual = FALSE;
    }
}

if ( dcl_function )
{
    if ( dcl_virtual )
    {
        if ( dcl_base == DESTRUCTOR_TYPE )
        {
            destructor_is_virtual = TRUE;
        }
        else
        {
            class_has_virtual_function = TRUE;
        }
    }
}

if ( tag_end )
{
    if ( tag_global && (tag_kind == CLASS_TAG) )
    {
        if ( tag_has_destr )
        {

```

```

        if ( class_has_virtual_function )
            {
                if ( ! destructor_is_virtual )
                    {
                        warn( 9255, "Destructor %s::~~%s() must be virtual.",
                            class_name(), class_name() );
                    }
            }
        else
            {
                warn( 9254, "Class %s needs a destructor.", class_name() );
            }
    }
}

```

// ----- Rules 2.6.1 and 2.6.2 -----

```

if ( dcl_friend )
    {
        if ( dcl_function )
            {
                if ( prefix("operator") )
                    {
                        if ( strequiv(root(), "=") )
                            ;
                        else if ( strequiv(root(), "+") )
                            ;
                        else if ( strequiv(root(), "-") )
                            ;
                        else if ( strequiv(root(), "*") )
                            ;
                        else if ( strequiv(root(), "/") )
                            ;
                        else if ( strequiv(root(), "%") )
                            ;
                        else if ( strequiv(root(), ">>") )
                            ;
                        else if ( strequiv(root(), "<<") )
                            ;
                        else
                            warn( 9261, "%s should not be a friend.", dcl_name() );
                    }
                else
                    warn( 9261, "Do NOT declare friend functions." );
            }
        else
            {
                warn( 9262, "Do NOT declare friend classes." );
            }
    }
}

```

```

// ----- Rule 2.7.1 -----

if ( keyword("class") )
    {
        detect_virtual_base = TRUE;
    }

if ( keyword("virtual") )
    {
        if ( detect_virtual_base )
            {
                warn( 9271, "Do NOT use virtual base classes." );
            }
    }

if ( dcl_base == CLASS_TYPE )
    {
        detect_virtual_base = FALSE;
    }

// ----- Rule 2.8.1 -----

if ( dcl_member )
    {
        if ( dcl_variable && dcl_base != DEFINED_TYPE )
            {
                warn( 9281, "Declare %s using a typedef name, not a basic C type.",
                    dcl_name() );
            }
    }

// ----- Rule 2.8.2 -----

if ( dcl_typedef )
    {
        if ( dcl_member == 0 )
            {
                warn( 9282, "Define typedef name %s in a base class.",
                    dcl_name() );
            }
    }

// ----- Rule 2.9.1 -----

// This rule is not currently enforceable with CodeCheck.

// ----- Rule 2.9.2 -----

```

```

// This rule is redundant (covered by 3.1.2).

// ----- Rule 2.9.3 -----

if ( tag_kind == ENUM_TAG )
{
    if ( tag_global )
    {
        warn( 9293, "Define enum %s in a base class.", tag_name() );
    }
}

// ----- Rule 3.1.1 -----

if ( dcl_parameter )
{
    is_object = ((dcl_base == CLASS_TYPE) || (dcl_base == STRUCT_TYPE));
    if ( is_object )
    {
        if ( (dcl_levels == 0) || ((dcl_levels == 1) && (dcl_level(0) ==
POINTER)) )
        {
            if ( dcl_abstract )
            {
                warn( 9311, "Declare parameter #%d to be a reference to %s.",
                    dcl_parameter, dcl_base_name() );
            }
            else
            {
                warn( 9311, "Declare parameter %s to be a reference to %s",
                    dcl_name(), dcl_base_name() );
            }
        }
    }
}

// ----- Rules 3.1.2 and 3.1.4 -----

if ( dcl_function )
{
    is_object = ((dcl_base == CLASS_TYPE) || (dcl_base == STRUCT_TYPE));
    if ( dcl_member && is_object )
    {
        if ( (dcl_levels == 1) || ((dcl_levels == 2) && (dcl_level(1) ==
REFERENCE)) )
        {
            if ( prefix("operator") )
            {
                if ( strequiv(root(), "=") )
                ;
            }
        }
    }
}

```

```

        else if ( strequiv(root(), "+") )
            ;
        else if ( strequiv(root(), "-") )
            ;
        else if ( strequiv(root(), "*") )
            ;
        else if ( strequiv(root(), "/" ) )
            ;
        else if ( strequiv(root(), "%") )
            ;
        else if ( strequiv(root(), ">>") )
            ;
        else if ( strequiv(root(), "<<") )
            ;
        else
        {
            warn( 9312, "%s::~%s() should not return an object.",
                class_name(), dcl_name() );
        }
    }
else
{
    warn( 9314, "Function %s::~%s() should not return an object.",
        class_name(), dcl_name() );
}
}
}
}

```

```
// ----- Rule 3.1.3 -----
```

```
// This rule cannot be enforced with this version of CodeCheck
```

```
// ----- Rule 3.1.5 -----
```

```

if ( dcl_parameter )
{
    if ( dcl_parameter == 1 )
    {
        non_ref_parm_found = FALSE;
    }

    if ( dcl_level(0) == REFERENCE )
    {
        if ( non_ref_parm_found )
        {
            warn( 9315, "Reference parameters must come first." );
        }
    }
    else // a non-reference fcn parameter has been found
    {
        non_ref_parm_found = TRUE;
    }
}

```

```

    }
}

// ----- Rule 3.1.6 -----

if ( dcl_function )
{
    is_constant = (dcl_level_flags(0) & CONST_FLAG);
    if ( dcl_member && is_constant )
    {
        warn( 9316, "Constant member functions should be avoided." );
    }
}

// ----- Rule 3.1.7 -----

if ( dcl_parameter )
{
    is_object = ((dcl_base == CLASS_TYPE) || (dcl_base == STRUCT_TYPE));
    if ( is_object && (dcl_levels == 1) )
    {
        is_constant = (dcl_level_flags(1) & CONST_FLAG);
        level = dcl_level( 0 );
        if ( is_constant && ((level == POINTER) || (level == REFERENCE)) )
        {
            if ( dcl_abstract )
            {
                warn( 9317, "Parameter #%d should not be const.",
                    dcl_parameter );
            }
            else
            {
                warn( 9317, "Parameter %s should not be const.", dcl_name() );
            }
        }
    }
}

// ----- Rule 4.1.1 -----

if ( lex_c_comment )
{
    warn( 9411, "Use /\ / comments, not \\/*...*/ comments." );
}

// ----- Rule 4.2.1 -----

if ( pp_const )

```

```

    {
    if ( lin_header && ! define_needed )
        {
        warn( 9421, "Declare %s as a const, not a macro.", pp_name() );
        }
    }

// ----- Rule 4.2.2 -----

if ( tag_anonymous )
    {
    if ( tag_global && tag_kind == ENUM_TAG )
        {
        warn( 9422, "This enumeration needs an enum type name." );
        }
    }

// ----- Rule 4.3.1 -----

if ( dcl_variable )
    {
    is_constant = (dcl_level_flags(dcl_levels) & CONST_FLAG);
    if ( dcl_global && is_constant )
        {
        warn( 9431, "Global constant %s should be a class member.",
            dcl_name() );
        }
    }

// ----- Rule 4.4.1 -----

if ( dcl_no_specifier )
    {
    if ( dcl_function )
        {
        warn( 9441, "Function %s needs an explicit return type.",
            dcl_name() );
        }
    }
}

```

# Chapter 8: Supporting Material

## 8.1 Glossary

abstract declarator	A declarator without an identifier. Abstract declarators are only used in two situations in C: in casts, and as the argument for <code>sizeof</code> .
ANSI	American National Standards Institute.
ASCII	American Standard Code for Information Interchange. A 7-bit coding scheme for the binary representation of alphabetic, numeric, and page-formatting information. ASCII is used by almost all computers except IBM mainframes and their compatibles, which use EBCDIC (although IBM microcomputers and their compatibles do not).
big endian	A computer is “big endian” (as opposed to “little endian”) if the low-order (“end”) byte in a word has a <i>larger</i> address than the high-order byte. The Motorola 68000 family is big endian, while the VAX and Intel 80x86 families are not. Synonym: left-to-right.
bitwise	Certain operators in C act in a “bitwise” fashion, meaning that the operator is separately applied to each bit of the operands.
BNF	Either “Backus-Naur Form” or “Backus Normal Form”, depending on how much credit you want to give to Dr. P. Naur. This is a language designed for use in describing language grammars.
cast	A “cast” is an explicit type conversion. In C, a cast consists of an abstract declarator surrounded by parentheses.
conjunction	The logical ( <i>i.e.</i> not bitwise) “and” operation.
declarator	A type specification. A declarator may or may not include an identifier. For example, the cast ( <code>char *</code> ) is an ab-

struct declarator without an identifier, while the expression `char *x` is a declarator with an identifier (*i.e.* `x`).

dereference	A pointer is “dereferenced” when the value is obtained from the address to which the pointer points. The origin of this awkward neologism is obscure, but it seems to have been invented by analogy with the word “reference”: if a variable can be <i>referenced</i> by a pointer, then the pointer must be <i>dereferenced</i> to obtain the value to which it points. It’s not English, but it computes.
disjunction	The logical ( <i>i.e.</i> not bitwise) “or” operation.
EBCDIC	Extended binary coded decimal interchange code. An 8-bit coding scheme for the binary representation of alphabetic, numeric, and page-formatting information. EBCDIC was invented by IBM for use by its mainframe computers. Pronounced “ <b>eb</b> sidick”.
enumeration tag	The name given to an enumeration. Once an enumeration has been tagged, it can be identified by tag when declaring other variables that refer to the same enumeration.
escape sequence	The C language permits the use of certain codes that stand for special characters ( <i>e.g.</i> <code>\b</code> is the C escape sequence for the backspace control character).
exception	An unexpected condition that the program encounters and cannot cope with. See C++ keywords <code>try</code> , <code>catch</code> , and <code>throw</code> .
extent	The “extent” of a variable is the time during which it refers to storage. Variables with static extent have storage allocated throughout execution of the program, while variables of local extent are allocated storage only while execution proceeds through the block in which they are declared. See “scope” and “visibility”.
hexadecimal	Base 16 numbers. The digits are: {0, 1, ... 8, 9, a, b, c, d, e, f}. Hexadecimal constants in C are indicated by the prefix “0x”.
identifier	A name for a variable or function.

indirection	The “indirection” operator (*) dereferences a pointer. That is to say, it yields the value stored in the address to which the pointer points.
infix operator	A binary operator is “infix” if it is written <i>between</i> its operands. For example, the logical <b>and</b> operator (&&) is infix. See also “prefix” and “postfix”.
initializer	A declaration in the C language may include an initial value for the variable that is declared. This value is its “initializer”.
ISO	International Standards Organization.
lexical analysis	A compiler is performing “lexical analysis” of source code when it is reading characters and collecting them into tokens ( <i>e.g.</i> names, numbers, strings, operators).
linkage	An identifier has “external linkage” if it is supposed to refer to the same object in each module in which it is declared. An identifier has “internal linkage” if it refers to a different object in each module.
lint	This is the name given in the early days of C to a utility program that performed basic error-checking ( <i>i.e.</i> lint-picking) on C source files. As C compilers have evolved sophisticated error-checking abilities of their own, lint programs have either become obsolete or have evolved into specialized error-checking niches.
little endian	A computer is “little endian” (as opposed to “big endian”) if the low-order (“end”) byte in a word has a <i>smaller</i> address than the high-order byte. The VAX and Intel 80x86 families are little endian, while the Motorola 68000 family is not. Synonym: right-to-left.
lvalue	An “lvalue” (pronounced <i>el-value</i> ) is an expression that refers to a region of memory that can be examined or altered. So-called because it is a value that can appear on the <i>left</i> side of an assignment.
macro	A macro is an expression that, in form, resembles a constant or a function call. However, the C preprocessor re-

places every macro expression with its appropriate C expansion *before compilation*.

manifest constant	A constant (in any computer language) is “manifest” if (a) its meaning is clearly apparent, and (b) its value never changes. Manifest constants are best created in C with the #define preprocessor directive.
module	The term “module” in C usually refers to a source file and all of its associated include files. Each module of a C program is compiled independently.
nil	Empty. Frequently used by former Pascal programmers as a synonym for “null”.
null pointer	A pointer whose value is not the address of any object. By convention, the address stored in a null pointer is zero. The fact that such a pointer actually points to address 0 is purely coincidental. Beware: in some machines the null pointer is <i>not</i> zero.
null statement	C permits a statement to be empty — the null statement consists of a single semicolon.
null string	An empty character string.
octal	Base 8 numbers. The digits are: {0, 1, 2, 3, 4, 5, 6, 7}. In older versions of C the digits 8 and 9 are also allowed, for reasons that surpass understanding. <b>Warning:</b> octal constants in C are indicated by the prefix digit “0”. Thus 010 is the same number as 8 (decimal).
overflow	The result of an arithmetic operation is said to “overflow” when the result is too large to represent using the specified type.
overload	An identifier has been “overloaded” if it is used for more than one purpose within a single block. Overloading is permitted (but not recommended) in certain specific contexts in C. In the worst case a single name can simultaneously refer to a macro, a statement label, a structure tag, a component name, and a variable name. In C++ the overloading is carried to even greater extremes.

PCC	The Portable C Compiler. This C compiler from Bell Labs implemented a version of the K&R standard, and served as the starting point for many commercial C compilers.
pointer	A variable whose value is an address.
postfix operator	A unary operator is “postfix” if it is written <i>after</i> its operand. The subscript selection operator (square brackets) is postfix. See also “prefix” and “infix”.
pragma	ANSI C allows the new preprocessor directive <code>#pragma</code> , which is used to supply implementation-defined information to the compiler.
prefix operator	A unary operator is “prefix” if it is written before its operand. The negation operator (-) is prefix. See also “postfix” and “infix”.
preprocessor	Unlike almost all other high-level languages, the C language has a built-in preprocessor which manipulates the source code in advance of actual compilation. The C preprocessor is primarily used for (a) expanding macro expressions into true C code, (b) inserting other files into the compilation stream, (c) controlling conditional inclusion of code, and (d) perpetrating sly programming tricks on poor unsuspecting compilers.
prototype	A new and welcome addition to the C language, a function prototype is a declaration which specifies the type of each of its formal parameters, and the type of its return value.
pseudo-unsigned	C compilers may treat the type <code>char</code> as either signed or unsigned. Unfortunately, compilers may also treat this type as “pseudo-unsigned”, meaning that its value is never negative but it is treated as though it were signed during type conversions. Ugly.
rule	If-statements are called “rules” in some computer languages (and especially in expert systems) if they satisfy these conditions: (a) the rules are not part of the program ( <i>i.e.</i> they are data, not code), and (b) the rules are not ordered.

scope	The “scope” of a declaration is the set of statements throughout which the declaration is active. Note: a declaration may not be “visible” within all statements of its scope. See “visibility” and “extent”.
semantics	A compiler is performing semantic processing when it is attaching meaning to the grammatical forms that are detected during syntactic analysis.
signal	A “signal” is an asynchronous event that requires special handling. Signals may be “raised” by the computer’s error detection mechanism, by the program itself, or by events external to the program.
signal handler	A “signal handler” is a C function that is invoked when its signal is “raised”. For example, an arithmetic overflow handler is a function that might allow a graceful exit to occur from a numerical computation when an overflow occurs.
storage class	A declaration can be modified by giving it a storage class specifier. The C storage classes are: <code>auto</code> , <code>extern</code> , <code>register</code> , <code>static</code> , and <code>typedef</code> . In C++ <code>typedef</code> is not a storage class.
syntactics	A compiler is performing syntactic analysis when it is reading a sequence of tokens and parsing them into grammatical patterns. Syntactic analysis follows lexical analysis, and precedes semantics.
token	The smallest unit of text recognized by a compiler, <i>e.g.</i> the keyword “ <b>else</b> ”, the operator “+=”, and the semicolon “;”.
trigraph	For the sake of keyboard devices that do not have the full complement of C characters, the ANSI standard allows the use of “trigraphs” to represent the missing characters. Each trigraph is a three-character sequence beginning with “??”.
unary	An operation is “unary” if it operates on exactly one operand.
underflow	The result of an arithmetic operation is said to “underflow” when the result is too small to represent using the specified type.

visibility	A declaration for an identifier is “visible” within a C program if the identifier is still associated with the declaration. Visibility is lost when the declaration is hidden by another declaration of the identifier in an inner block. See “scope” and “extent”.
void	The absence of a quantity ( <i>not</i> the quantity zero). A function which returns “void” is a function which returns no value at all. A “pointer to void” is the generic undefined pointer to data in ANSI C.
volatile	In ANSI C, a variable may be declared “volatile”, meaning that it must be treated with great care by optimizing compilers.
whitespace	Source characters that consist of blank, tab, and return characters. In some contexts a comment also counts as whitespace.
xor	The exclusive “or” operation.
yacc	“Yet Another Compiler Compiler”. This is a program that takes as input a BNF description of a language grammar and produces as output a parser for that language.

## 8.2 Bibliography

- AN90 X3J11 Technical Committee (1990) **American National Standard for the Programming Language C**. American National Standards Institute.
- AN95 Doc No:X3J16/95-0087 (1995) **Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++** CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005
- AN96 Doc No:X3J16/96-0225 (1996) **Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++**. ITIC, 1250 Eye Street NW, Suite 200, Washington DC 20005
- CDS86 Conte, SD, Dunsmore, HE, and Shen, VY (1986) **Software Engineering Metrics and Models**. Benjamin.
- DM87 DeMillo, RA, McCracken, WM, Martin, RJ and Passafiume, JF (1987) **Software Testing and Evaluation**. Benjamin/Cummings.
- GS96 Glass, G and Schuchert B(1996) **The STL <Primier>**. Prentice Hall.
- GC87 Grady, RB and Caswell, DL (1987) **Software Metrics: Establishing a Company-Wide Program**. Prentice-Hall.
- ES90 Ellis, MA, and Stroustrup, B (1990) **The Annotated C++ Reference Manual**. Addison-Wesly.
- POSIX.1 IEEE Technical Committee on Operating Systems (1990) **Information Technology — Portable Operating System Interface — Part 1: System Application Program Interface (API) [C Language]**. ISO/ IEC 9945-1; IEEE Std 1003.1-1990. Institute of Electrical and Electronic Engineers.
- MH77 Halstead, MH (1977) **Elements of Software Science**. Elsevier.
- HS84 Harbison, SP and Steele, GL (1984) **C: A Reference Manual**. Prentice-Hall.
- HS88 Harbison, SP and Steele, GL (1987) **C: A Reference Manual, 2nd Edition**. Prentice-Hall.

- HS91 Harbison, SP and Steele, GL (1991) **C: A Reference Manual, 3rd Edition**. Prentice-Hall.
- MRH90 Horton, MR (1990) **Portable C Software**. Prentice-Hall.
- RJ88 Jaeschke, R (1988) **Portability and the C Language**. Hayden Books.
- KR78 Kernighan, BW and Ritchie, DM (1978) **The C Programming Language**. Prentice-Hall.
- KR88 Kernighan, BW and Ritchie, DM (1988) **The C Programming Language, 2nd Edition**. Prentice-Hall.
- AK89 Koenig, A (1989) **C Traps and Pitfalls**. Addison-Wesley.
- OL86 Lecarme, O and Gart, ML (1986) **Software Portability**. McGraw-Hill.
- MSD92 Meyers, SD(1992) **Effective C++**. Addison-Wesley.
- MSD96 Meyers, SD(1996) **More Effective C++**. Addison-Wesley.
- OS95 ObjectSpace(1995) **STL<ToolKit>**. ObjectSpace
- PS81 Perlis, A, Sayward, F and Shaw, M (1981) **Software Metrics: An Analysis and Evaluation**. MIT Press.
- TP84 Plum, T (1984) **C Programming Guidelines**. Prentice-Hall.
- TP89 Plum, T (1989) **C Programming Guidelines, 2nd Edition**. Prentice-Hall.
- RC90 Rabinowitz, H and Chaim, S (1990) **Portable C**. Prentice-Hall.
- BS91 Stroustrup, B (1991) **The C++ Programming Language, 2nd Edition**. Addison-Wesley.
- BS97 Stroustrup, B (1997) **The C++ Programming Language, 3rd Edition**. Addison-Wesley.
- VW88 Vincent, J, Waters, A, and Sinclair, J (1988) **Software Quality Assurance, Volume 1**. Prentice-Hall.

## 8.3 Index

- "-D, 22
- analysis
  - lexical, 150
  - semantic, 153
  - syntactic, 153
- ANSI, 148
- array initializer, 48
- ASCII, 148
- assembler, 20, 36
- AT&T C++, 4
- backslash, 37
- binary decision point*, 99
- bitfield, 49, 50
- bitwise, 148
- BNF, 148
- boolean, 68
- Borland, 4, 84
- braces, 70
- C++, i, v, 1, 4, 5, 9, 10, 11, 16, 19, 21, 22, 24, 38, 47, 55, 66, 68, 75, 76, 83, 84, 92, 93, 95, 133, 135, 149, 151, 153, 155, 156
  - dialects, 4
  - example rules, 133
- cast, 148
- CCEXCLUDE, 6
- CCRULES, 2, 10
- characters
  - constants, 32
  - nonstandard, 30
  - pseudo-unsigned, 49, 152
  - standard, 30
  - trigraph, 31
  - underscore, 29
  - wide, 38
- cnv\_any\_to\_ptr, 28
- cnv\_ptr\_to\_ptr, 28
- CodeCheck
  - predefined constants, 20
- CODECHECK, 20
- CodeWarrior, 4
- comma, 69
  - operator, 105, 110, 111
  - separator, 69
- command line, 1
- comment
  - //, 75
  - macro, 42
  - nested, 4, 75
- Compiler Drift, 14
- complexity, 99
- Concatenation, 36
- conjunction, 148
- const, 84
- constants
  - hexadecimal, 35
  - manifest, 32, 67, 151
  - octal, 35
  - predefined, 20
- Continuation, 37
- dcl\_3dots*, 52
- dcl\_all\_upper*, 62
- dcl\_any\_upper*, 29
- dcl\_auto\_init*, 48
- dcl\_base*, 25, 49, 50, 54, 65
- dcl\_base\_name()*, 26, 65
- dcl\_bitfield*, 50
- dcl\_bitfield\_anon*, 50
- dcl\_bitfield\_arith*, 50
- dcl\_bitfield\_size*, 50
- dcl\_count*, 51
- dcl\_cv\_modifier*, 84
- dcl\_empty*, 50
- dcl\_enum\_hidden*, 62
- dcl\_explicit*, 85
- dcl\_extern*, 59, 65, 83
- dcl\_extern\_ambig*, 30, 59
- dcl\_first\_upper*, 62
- dcl\_from\_macro*, 65
- dcl\_function*, 52, 53, 54
- dcl\_global*, 26, 59, 65, 83
- dcl\_hidden*, 62, 65
- dcl\_Hungarian*, 66
- dcl\_ident\_length*, 59, 81
- dcl\_init\_arith*, 49
- dcl\_initializer*, 54, 83
- dcl\_label\_overload*, 62
- dcl\_level()*, 25, 52
- dcl\_level\_flags()*, 26

- dcl\_levels, 25
- dcl\_local, 26, 59, 66
- dcl\_name(), 25
- dcl\_need\_3dots, 52
- dcl\_no\_specifier, 82
- dcl\_not\_declared, 82
- dcl\_oldstyle, 52
- dcl\_parm\_hidden, 57
- dcl\_signed, 66
- dcl\_simple, 54
- dcl\_static, 54, 59, 66
- dcl\_template, 83
- dcl\_typedef, 53, 63, 66
- dcl\_typedef\_dup, 53
- dcl\_underscore, 30, 34
- dcl\_union\_bits, 50
- dcl\_union\_init, 48
- dcl\_unsigned, 66
- dcl\_variable, 83
- decisions, 99
- declarator, 148
  - abstract, 148
- default.cco, 2, 5
- defined, 43, 45
- dereference, 149
- dialect, 4
- disjunction, 149
- EBCDIC, 149
- elif, 43
- endian
  - big, 148
  - little, 150
- end-of-file, 36
- ENUM\_TYPE, 49
- enumerated constant, 49
- environment
  - CCEXCLUDE, 1
  - CCRULES, 1
  - CIncludes, 1
  - INCLUDE, 1
- escape, 149
- escape codes, 31
- Escape Sequences, 32
- exception, 149
- exception handler, 93
- exp\_not\_ansi, 55
- exp\_operands, 108
- exp\_operators, 107
- exp\_tokens, 108

- explicit, 85
- extensions
  - Borland, 4, 84
  - C++, 4
  - CodeCheck, 8
  - HP/Apollo, 4
  - Intel, 84
  - Metaware, 4, 45, 84
  - Microsoft, 4, 51, 84
  - MPW C, 51
  - Symantec, 4
  - Think C, 51
  - Vax, 4, 45
- extent, 149
- extern**, 59, 65, 83, 153
- far, 63, 84
- fcn\_com\_lines, 90
- fcn\_decisions, 100
- fcn\_exec\_lines, 90
- fcn\_H\_operators, 96
- fcn\_high, 94
- fcn\_low, 94
- fcn\_nonexec, 94
- fcn\_operands, 96, 108
- fcn\_operators, 96, 108
- fcn\_tokens, 96, 108
- fcn\_total\_lines, 90
- fcn\_u\_operands, 96
- fcn\_uH\_operators, 96
- fcn\_white\_lines, 90
- file
  - listing, 4, 6
  - object, 2
  - project, 2
  - prototypes, 6
  - rule, 2, 5
  - stderr.out, 4
- flowchart, 100
- flowgraph, 100
- Gnu C, 55
- Hall, Mark R., 156
- Halstead, Maurice, 95
- Harbison, S.P., 155, 156
- header files
  - search path, 3
  - suppress checking, 6
- hexadecimal, 149
- huge, 84
- Hungarian prefixes, 63

- identifier, 149
- identifier()*, 37
- idn\_global, 26
- idn\_local, 26
- idn\_member, 26
- idn\_parameter, 26
- indentation, 70
- indirection, 150
- infix, 150
- initializer, 150
- initializers
  - array, 48
- Installation, 1
- Intel, 84
- ISO, 16, 150
- Jaeschke, Rex, 156
- Kernighan, B.W., 156
- keyword()*, 37, 52
- Koenig, Andrew, 13, 156
- labels, 83
  - lex\_ansi\_escape*, 33
  - lex\_assembler*, 37
  - lex\_backslash*, 37
  - lex\_bad\_call*, 41
  - lex\_big\_octal*, 33
  - lex\_char\_empty*, 33
  - lex\_char\_long*, 33
  - lex\_cpp\_comment*, 75
  - lex\_float*, 35
  - lex\_hex\_escape*, 32, 33
  - lex\_initializer*, 68, 76
  - lex\_invisible*, 38
  - lex\_lc\_long*, 74
  - lex\_long\_float*, 35
  - lex\_nl\_eof*, 36
  - lex\_nonstandard*, 30
  - lex\_not\_KR\_escape*, 33
  - lex\_not\_manifest*, 67, 76
  - lex\_null\_arg*, 41
  - lex\_num\_escape*, 32, 77
  - lex\_punct\_after*, 69
  - lex\_punct\_before*, 69
  - lex\_radix*, 35
  - lex\_str\_concat*, 36
  - lex\_str\_length*, 59
  - lex\_str\_macro*, 44
  - lex\_str\_trigraph*, 31
  - lex\_suffix*, 35, 74
  - lex\_trigraph*, 31
  - lex\_unsigned*, 35

- lex\_wide*, 38
- lex\_zero\_escape*, 32
- lexical, 150
- lexical guidelines, 29
  - lin\_continuation*, 71
  - lin\_continues*, 69
  - lin\_dcl\_count*, 82, 89
  - lin\_end*, 89
  - lin\_has\_code*, 89, 108
  - lin\_has\_comment*, 82, 89
  - lin\_header*, 89
  - lin\_include\_kind*, 89
  - lin\_include\_name()*, 89
  - lin\_indent\_space*, 71
  - lin\_indent\_tab*, 71
  - lin\_is\_comment*, 89
  - lin\_is\_exec*, 89, 108
  - lin\_is\_white*, 89
  - lin\_length*, 59
  - lin\_nest\_level*, 2, 71
  - lin\_nested\_comment*, 75
  - lin\_number*, 90
  - lin\_operands*, 96, 108
  - lin\_operators*, 96, 108
  - lin\_preprocessor*, 89
  - lin\_source*, 90
  - lin\_suppressed*, 89
  - lin\_tokens*, 96, 108
- linkage, 150
- lint, v, 150
- logic structure, 100
- long float, 34
- lvalue, 150
- machismo, 13
- macro, 150
- macros
  - predefined, 20
- maintainability, 99
- manifest constants, 67
- McCabe, 100
- Metaware, 4, 84
- Metrowerks, 4
- Microsoft, 4, 51, 84
  - mod\_com\_lines*, 90
  - mod\_decisions*, 100
  - mod\_exec\_lines*, 90
  - mod\_functions*, 98
  - mod\_H\_operators*, 97
  - mod\_high*, 94

*mod\_low*, 94  
*mod\_macros*, 98  
*mod\_nonexec*, 94  
*mod\_operands*, 97, 109  
*mod\_operators*, 97, 109  
*mod\_tokens*, 97, 109  
*mod\_total\_lines*, 90  
*mod\_u\_operands*, 97  
*mod\_uH\_operators*, 97  
*mod\_white\_lines*, 90  
 module, 151  
 MPW C, 51  
     typedef names, 53  
*mutable*, 85  
 near, 63, 84  
 nested  
     comments, 75  
     header files, 59  
     logic, 103  
 nested.: if directives  
 newline, 36  
 NUXI, 33  
 octal, 151  
*op\_base()*, 27  
*op\_cast*, 27  
*op\_declarator*, 69  
*op\_executable*, 69  
*op\_high*, 69  
*op\_infix*, 69  
*op\_level()*, 27  
*op\_level\_flags()*, 27  
*op\_levels()*, 27  
*op\_low*, 69  
*op\_medium*, 69  
*op\_operands*, 27  
*op\_postfix*, 69  
*op\_prefix*, 69  
*op\_space\_after*, 70  
*op\_space\_before*, 70  
*op\_white\_after*, 70  
*op\_white\_before*, 70  
 options  
     **-A**, 2  
     **-B**, 2, 70, 71  
     **-C**, 2  
     **-D**, 2  
     **-E**, 3  
     embedded SQL, 6  
     **-F**, 3

**-G**, 3  
     **-H**, 3  
     **-I**, 3  
     **-J**, 3  
     **-K**, 4, 42  
     **-L**, 4  
     **-M**, 4  
     macros, 4  
     **-N**, 4  
     **-NEST**, 4  
     nested classes, 4  
     nested comments, 4  
     **-O**, 4  
     **-P**, 5  
     progress, 5  
     prototypes, 6  
     **-Q**, 5  
     **-R**, 5  
     rule file, 5  
     **-S**, 5  
     **-SQL**, 6  
     stderr.out, 4  
     **-T**, 6  
     **-U**, 6  
     user defined, 6  
     **-V**, 6  
     variables, 7  
     **-W**, 6  
     **-X**, 6  
     **-Y**, 6  
     **-Z**, 6  
 options.: include files  
 overload, 151  
 Parochialism, 13  
 pascal, 84  
     Microsoft C, 51  
     MPW C, 51  
     Think C, 51  
     type modifier, 51  
     type specifier, 51  
 PCC, 151  
 Plum,Thomas, 29, 156  
 pointer, 152  
     null, 151  
 portation problem  
     source, 16  
     target, 16  
 postfix, 152

- pp\_ansi*, 45
- pp\_arg\_count*, 41, 60
- pp\_arg\_paren*, 78
- pp\_arg\_string*, 44
- pp\_arith*, 41
- pp\_assign*, 78
- pp\_bad\_white*, 39
- pp\_benign*, 79
- pp\_comment*, 42
- pp\_defined*, 43, 45
- pp\_depend*, 80
- pp\_elif*, 43, 45
- pp\_empty\_arglist*, 41
- pp\_error*, 45
- pp\_if\_depth*, 60
- pp\_include*, 45
- pp\_include\_depth*, 60
- pp\_include\_white*, 40
- pp\_length*, 47
- pp\_lowercase*, 68
- pp\_macro\_conflict*, 68, 86
- pp\_macro\_dup*, 68, 86
- pp\_overload*, 42
- pp\_paste*, 42, 45
- pp\_pragma*, 45
- pp\_recursive*, 46
- pp\_relative*, 47
- pp\_semicolon*, 78
- pp\_sizeof*, 41
- pp\_stack*, 68, 80
- pp\_stringize*, 45
- pp\_sub\_keyword*, 46
- pp\_trailer*, 78
- pp\_undef*, 68, 80
- pp\_unknown*, 45
- pp\_unstack*, 80
- pp\_white\_after*, 39
- pp\_white\_before*, 39
- pragma, 152
- predefined macros, 20
- prefix*, 66, 152
- preprocessor, 152
  - argument, 44, 78
  - arguments, 41, 60
  - arithmetic, 40
  - define, 68, 80
  - defined, 43, 45
  - elif, 43
  - if, 40
  - keywords, 46
  - semicolon, 78
  - sizeof, 40, 41
  - undef, 68, 80
  - whitespace, 39, 40
- preprocessor.: elif
  - prj\_com\_lines*, 91
  - prj\_conflicts*, 86
  - prj\_decisions*, 100
  - prj\_exec\_lines*, 91
  - prj\_functions*, 98
  - prj\_H\_operators*, 97
  - prj\_high*, 94
  - prj\_low*, 94
  - prj\_macros*, 98
  - prj\_nonexec*, 94
  - prj\_operands*, 97, 109
  - prj\_operators*, 97, 109
  - prj\_tokens*, 97, 109
  - prj\_total\_lines*, 91
  - prj\_u\_operands*, 97
  - prj\_uH\_operators*, 97
  - prj\_white\_lines*, 91
- project, 2
- prototype, 152
- prototypes
  - creation, 6
  - recursive, 46
  - reserved keywords:, 37
  - Ritchie, D.M., 156
  - rule, 152
  - scope, 152
  - semantics, 153
  - signal, 153
  - signal handler, 153
  - signed, 66
  - sizeof, 40
  - specifier
    - storage class, 51, 153
    - type, 51, 84
- SQL, 6
- standard
  - K&R, 15
  - PCC, 15
- statement
  - null, 151
- Steele, G.L., 155, 156
- stm\_bad\_label*, 56
- stm\_cases*, 92
- stm\_catches*, 93
- stm\_cp\_begin*, 73, 93

*stm\_depth*, 103  
*stm\_end*, 93  
*stm\_end\_tryblock*, 93  
*stm\_is\_comp*, 74, 93  
*stm\_is\_expr*, 93  
*stm\_is\_high*, 93  
*stm\_is\_iter*, 93  
*stm\_is\_jump*, 93  
*stm\_is\_low*, 93  
*stm\_is\_nonexec*, 93  
*stm\_is\_select*, 93  
*stm\_labels*, 93  
*stm\_lines*, 90  
*stm\_never\_caught*, 93  
*stm\_operands*, 96, 108  
*stm\_operators*, 96, 108  
*stm\_tokens*, 96, 108  
storage class, 153  
storage classes  
    pascal, 51  
string  
    null, 151  
    wide, 38  
style, 11  
suffix, 66  
    F, 34  
    L, 34  
    U, 34  
Symantec, 4  
syntax, 153  
System Variables, 34  
tab stops, 70  
tag  
    enumeration, 149  
Think C, 51  
token, 95, 153  
trigraph, 31, 153  
try blocks, 92  
type  
    modifier, 51, 84  
    specifier, 51, 84  
-U, 22  
unary, 153  
undef, 68, 80  
underflow, 153  
visibility, 153  
void, 154  
volatile, 84, 154  
whitespace, 39, 40, 154  
xor, 154  
yacc, 154