CodeCheck Technical Notes Spring 2006 Version 13.00

CodeCheck Technical Notes are technical discussions for those who are installing CodeCheck, writing CodeCheck rules, or writing scripts that invoke CodeCheck. These notes supplement and amplify upon material in ***The CodeCheck Reference Manual*** and ***C and C++ Source Code Analysis Using CodeCheck***, and provide important information that is particular to machines, compilers, libraries, and operating systems.

Table of Contents:

These technical notes are up-to-date for CodeCheck version 12.50. All rule files and support tips are now available via World Wide Web (**http://www.abraxas-software.com/dl**). All online versions of all documentation is available to customers (**http://www.abraxas-software.com/pdf**).

Using Abraxas Software site [`www.abraxas-software.com`]

Currently we have placed on our www site (www.abxsoft.com) all relevant CodeCheck documentation.

Change.txt	Log of all CodeCheck changes to date – Request by Email.
Master.txt	The complete list of all CodeCheck options, functions, and variables. Text format for easy searching.
Rule_idx.txt	A list of all CodeCheck rule files available from Abraxas Software.
Rules.zip	The standard Rule Files [.cc] currently available in ZIP format. Individual rule files may be downloaded from the site.

If you don't see what your looking for then please send Email to support@abxsoft.com and tell us what you what you need. We will Email it to you within 24 hours.

---

www.abxsoft.com/dl [/public.zip]

We have created a public rule file (/public.zip) on the site, all submissions must be sent via Email to support@abxsoft.com, please mention you would like to have your rule file posted in www.abxsoft.com/dl

• • • Important • • •

The latest editions of ***The CodeCheck Reference Manual*** and ***C and C++ Source Code Analysis Using CodeCheck*** are dated February 23, 2006. To order the new editions from Abraxas, please call 1-503-232-0540, or send Email to support@abraxas-software.com. To fax please use USA 503-232-0543. When submitting problem reports use form-format found on last page of this document, include materials requested.

#1 CodeCheck MS-DOS / WIN-3.x

Written by: Patrick Conley
Last Revised: 1 October 1998

This Technical Note describes the features and requirements of CodeCheck for DOS computers.

Which version is appropriate?

CodeCheck for the DOS operating system comes in several versions. The version of CodeCheck written for standard DOS, named `check.exe`, is limited by the severe memory restrictions imposed by DOS. In fact, since CodeCheck 5.0 we are no longer able to support 640K, i.e., the 16 bit world. Currently for CodeCheck to used on DOS/WIN 3.x we include a 32 Bit DOS Extended version called **chk32.exe**, which is based on technology from Borland International.

There are several ways to overcome these memory restrictions. If you wish to run CodeCheck within a DOS emulator under a different operating system, *e.g*. IBM OS/2 2.x, MS Windows NT, DEC VMS, Apple Macintosh, or any dialect of Unix, then Abraxas Software *strongly* recommends instead that you use the version of CodeCheck that was compiled specifically for your operating system. If, however, you have DOS running on a 386, 486, or Pentium computer with at least 8 megabytes of extended memory, then you can use `CHK32.exe`. This version of CodeCheck takes full advantage of all available extended memory, the 80386 processor's 32-bit instruction set, and 32-bit addressing. It is functionally equivalent to `check.exe`, but is not subject to the 640k memory limit imposed by DOS.

What it requires

To use `CHK32` you must have a 386 (or later) computer, running MS-DOS 5.0 or later, or DR-DOS 6.0 or later. `CHK32` functions well in systems with at least **16 megabytes of *available*** extended memory. However if you are analyzing MS-VC++ 4.0 or higher then 64 MB should be considered the minimum. If your checking MSDEV 5.0 or higher then 128MB RAM and use a 400MHZ Pentium II.

Compatibility

`CHK32` can execute while any of these memory managers are active:

- Helix NETROOM version 2.2 or higher
- Microsoft Windows 3.0 and 3.1 in enhanced mode
- Qualitas 386MAX version 6.0 or higher
- Quarterdeck QEMM386 version 6.00 or higher

`CHK32` can execute while any of these disk-cache utilities are active:

- Hyperware HyperDisk
- Microsoft SMARTDrive
- Multisoft Super PC-KWIK
- Qualitas QCache

`CHK32` is happiest in the DOS / WIN3.x environment when it has 16 megabytes or more of extended memory to use.

Microsoft Windows

CHK32 runs under Windows 3.x, Windows NT/95, and OS/2. It should be run from the MS-DOS prompt, not from the File Manager. If your operating system is Windows NT then please contact Abraxas for the 32-bit NT version of CodeCheck if you wish to support long file names.

If you wish to run CHK32 under Microsoft Windows, ***and if your computer does not have a math coprocessor***, then you will need to install a floating-point emulator. Your computer has no math coprocessor if it is a 386 without a 387 chip installed, or if it is a 486SX chip. Note: the 486DX, and Pentium chips do have on-board math coprocessors.

Some DOS and Windows environments may require the file **32rtm.exe** when running chk32.exe, please place 32rtm.exe and **dmpi32vm.ovl** in your system PATH. (\bin) These files can be found in the container file **32bit.zip** included on the CodeCheck DOS/WIN distribution disk. These files may be obtained from our ftp site at ftp.abxsoft.com.

Windows 3.x may require WINDPMI.386, if this file is not already installed then place it in c:\windows and add the following line to the file system.ini in c:\windows.

[386Enh]

device=c:\windows\windpmi.386

IBM OS/2 3.x

CHK32 will run in a DOS box under OS/2 version 2.0 or later if this operating system was installed with the DOS protected mode option, ***and*** if you have a math coprocessor. Be sure to change the settings for the DOS Full Screen Command Prompt so that at least 8 megabytes of extended memory are available.

MS-DOS and DR-DOS

If you use the DOS=HIGH option in CONFIG.SYS then you *must* use a third-party memory manager (*e.g*. Windows, QEMM-386, 386MAX, EMM386, or NETROOM) to use CHK32.

CHK32 can manage memory on its own, without the assistance of any third-party memory manager, but only if the DOS=HIGH option is ***not*** used in CONFIG.SYS.

CHK32 is ***not*** compatible with the Task Swapper in DOSSHELL of MS-DOS.

Quarterdeck QEMM-386

Do not use the NOXMS option. If you use the DOS=HIGH option in CONFIG.SYS, then do not use the QEMM-386 OFF option.

Qualitas 386MAX

If you use either the EMS=n option or the EXT=n option, set n to a value greater than 0 to enable services required by CHK32.

InterSolv PolyMake

If you wish to invoke CodeCheck under InterSolv PolyMake, be sure ***not*** to use the asterisk operation line modifier in front of the CodeCheck call:

`* CHK32 -Rmyrules` *(etc.)* *This asterisk (`*`) will cause a crash!*

Predefined macros

CodeCheck always predefines certain macros before it reads a single line of source code. Which macros are predefined, and which values they have, depend on which operating system is in use and which **–K** option is in effect. To determine exactly which macros are predefined, and their values, use the **–D?** command-line option. For example, the command `chk32 -K4 -D?` will cause CodeCheck to print the list of macros that are predefined when generic C++ source files are checked.

In addition to the standard ANSI predefined macro constants, the following macros are predefined when no **–K** option is specified. (**-D?**)

```
MSDOS              __MSDOS__          __I86__
M_I86              __LARGE__          __386__
M_I386             M_I86LM            _WIN32
```

If your compiler is from Borland, Microsoft, Symantec, or Watcom, then please see the corresponding Tech Note for compiler-specific instructions and macro definitions.

If necessary, any predefined macro may be undefined with the **`-U`** command-line option, or given a different value with the **–D** option. The special option **–D?** will cause CodeCheck to print a list of all predefined macros.

Insufficient Extended Memory?

`CHK32` will issue an "insufficient extended memory" message if it cannot obtain more dynamic memory from the operating system. The minimum amount of memory for using CodeCheck should be considered 16 Megabytes of RAM, 8 MB of RAM will only provide the ability to analyze very small C/C++ files and not entire projects.

#2 CodeCheck Unix

Written by: Patrick Conley
Last Revised: 22 June 2002

This Technical Note discusses the installation and execution of CodeCheck on Unix computers.

Shrouded sources

CodeCheck Unix is supplied as "shrouded" C source files. Shrouding is a process that renders the source code intelligible only to C compilers. Shrouding removes all formatting and comments, encodes all identifier names, and converts high-level grammatical constructs (*e.g.* `while` and `for`) into low-level code (`if` and `goto` statements). We distribute CodeCheck in shrouded sources in order to maximize portability while still maintaining security for our intellectual property.

CodeCheck requires an ANSI-C compiler

The source files for CodeCheck require an ANSI compiler. If your normal compiler is not ANSI-conforming, then there are several alternatives. First, your Unix installation may have Gnu C. This is an excellent shareware C compiler for Unix computers that has been very widely distributed around the world by the Free Software Foundation. Many Abraxas customers have compiled CodeCheck with the Gnu C compiler without trouble. Second, almost any C++ compiler will do as well, since C++ is based on ANSI C, *but its preprocessor must also be ANSI-conforming*.

The CodeCheck source code for UNIX is "shrouded" to maintain our proprietary technology and patented advantages. The shrouding may create some debugging problems. The shrouded source code is different than the original source code, especially with the names of user defined types. This may cause problems during linking with C++ compilers which are very sensitive. If you have difficulties compiling or linking the shrouded source code please contact send us an example of the error messages you are getting from your compiler and/or linker.

Special instructions for QNX

If your flavor of Unix is QNX, then you must define the macro `CC_QNX` when compiling CodeCheck. Do this by adding this option to `CFLAGS` in the makefile for compiling CodeCheck:

```
-dCC_QNX
```

If your "long" Integer Type is wider than 32 bits

The CodeCheck sources assume that the "long" integer type is 32 bits wide. If your long integer is wider than 32 bits (for example, on the new Alpha chip from DEC), then add this command-line option to the makefile for compiling CodeCheck:

```
-dLONG64
```

`If you have other special requirements please contact Abraxas Software.`

How CodeCheck searches for header files

CodeCheck looks for header files in the same way as virtually all Unix compilers. The default directory is `/usr/include`. If any command-line options **–I** have been used, then directories given in these options are searched before the default directory. If desired, the **–I** command-line options can be placed in project files, one per line.

CodeCheck also looks at the INCLUDE environmental variable for header directories. You may set this variable to identify the header directory paths. For example, if some of your C headers are located in /usr/zinc/include, then the following C-Shell command will set the INCLUDE variable to the correct path:

```
setenv INCLUDE "/usr/zinc/include"
```

If your headers are located in more than one directory, list all directory paths in this variable, separated by colons.

CodeCheck searches directories for system header files in this order:

1. Directories specified by calling the CodeCheck function set_str_option('I', …).
2. Directories specified with **-I** in the command-line or in project files.
3. Directories specified in the INCLUDE environmental variable.
4. The default header directory, /usr/include.
5. The current directory (the directory where the file issuing the #include directive is located).

How CodeCheck finds rule files

CodeCheck searches for rule files in the default directory /usr/CodeCheck/rules, and then in the current directory. Note the two upper-case C's in the name of this default directory.

CodeCheck also looks at the CCRULES environmental variable for rule directories. You may set this variable to identify the rule directory paths. For example, if some of your CodeCheck rule files are located in /usr/foobar, then the following C-Shell command will set the CCRULES variable to the correct path:

```
setenv CCRULES "/usr/foobar"
```

Directories listed in CCRULES are searched before the default rule directory. If your rule files are located in more than one directory, list all directory paths in this variable, separated by colons.

Predefined macros

CodeCheck always predefines certain macros before it reads a single line of source code. Which macros are predefined, and with which values, depend on which operating system is in use and which **–k** option is in effect. To determine exactly which macros are predefined, and their values, use the **–d?** command-line option. For example, check -k4 -d? will cause CodeCheck to print the list of macros that are predefined when generic C++ source files are checked.

The following macros are predefined when no **–k** option is specified. (-D?)

unix	**unix**	**builtin va alist**

If necessary, any predefined macro may be undefined with the **–U** command-line option, or given a different value with the **–D** option. The special option **–D?** will cause CodeCheck to print a list of all predefined macros.

Special HPUX, AIX, SUN, SGI, and SOLARIS Caveats

Extremely Important: most compilers have some predefined macros. Be sure to define yours on the command-line when invoking CodeCheck. Failure to do so will result in highly mysterious syntax errors whenever a header or source file contains one of these macro names. Here are some examples that we have learned about:

HPUX: You must define the macro **__hpux** on the command-line. On HP 9000 computers, you must define a macro on the command-line that indicates which CPU you have: **__hp9000s300, __hp9000s700**, *or* **__hp9000s800.**

AIX: You must define the macro **_AIX** on the command-line.

SUN: Either **sun3** or **sun4** must be defined on the command-line.

SGI: **_MIPS_SZLONG** must be defined 32 or 64 depending on architecture.

SOLARIS: Please define one of macros **__i386, i386, __ppc, __sparc**, or **sparc** for the machine type. Otherwise you may get error message "ISA not supported" from included header file "*isa_defs.h*" which requires one of macros above to be defined.

It is suggested to have the macro **__STDC__** defined on the command line if you are checking C++ code with system header files included. If this macro is not defined, it is possible `/**/` will be the paste operator used in some system header files, and `/**/` is effective as macro paste operator only when option -K0 or -K2 is used.

#3 CodeCheck OS/2

Written by: Patrick Conley
Last Revised: 15 December 1999

This Technical Note discusses the behavior of CodeCheck OS/2.

The CodeCheck OS/2 executable

The CodeCheck OS/2 executable is named `CHECK2.EXE`. It must be run in the OS/2 command-line environment, not in the DOS box. It must be run under OS/2 version 2.0 or later. It is not compatible with earlier 16-bit versions of OS/2.

Presentation Manager is not supported

CodeCheck OS/2 is designed for use with the OS/2 command-line interpreter, either full-screen or within a window. It does not provide a graphical user interface. Some compiler vendors allow CodeCheck to be built-in via an options pull-down, please see your compiler user manual if you require a GUI interface for CodeCheck/2.

Predefined macros *(Note: these macros changed 1-1-95)*

CodeCheck always predefines certain macros before it reads a single line of source code. Which macros are predefined, and which values they have, depend on which operating system is in use and which **–K** option is in effect. To determine exactly which macros are predefined, and their values, use the **–D?** command-line option. For example, the command `check -K4 -D?` will cause CodeCheck to print the list of macros that are predefined when generic C++ source files are checked.

In addition to the standard ANSI predefined macro constants, the following macros are predefined when no **–K** option is specified (these are the correct macros for the C-SET compiler):

```
__IBMC__    __32BIT__    _M_I386    __386__    __OS2__    __FLAT__
```

If you use the –K4 switch for C++, then these macros are also predefined:

```
__IBMCPP__    __cplusplus    c_plusplus
```

If necessary, any predefined macro may be undefined with the **`-U`** command-line option, or given a different value with the **–D** option.

See the Tech-Note section on IBM Visual Age C++ Compiler for latest information.

If you use the Borland compiler on OS/2

Very important: CodeCheck OS/2 assumes that your compiler is IBM C-SET. If you use the Borland C/C++ compiler, then you must define `__BORLANDC__` and `__TURBOC__` on the command-line when invoking CodeCheck. For example, to check a C++ file named foo.cpp, use:

```
check  foo.cpp  -k4  -d__BORLANDC__=0x0500  -d__TURBOC__=0x0500
```

#4 CodeCheck VMS

Written by: Loren Cobb
Last Revised: 09 April, 1996

This Technical Note discusses the installation and behavior of CodeCheck on computers with the VMS operating system from Digital Equipment Corporation.

Shrouded sources

CodeCheck VMS is supplied as "shrouded" C source files. Shrouding is a process that renders the source code intelligible only to C compilers. Shrouding removes all formatting and comments, encodes all identifier names, and converts high-level grammatical constructs (*e.g.* `while` and `for`) into low-level code (`if` and `goto` statements). We distribute CodeCheck in shrouded sources in order to maximize portability while still maintaining security for our intellectual property.

Define the `CCRULES` logical name

Before CodeCheck can successfully read your C source files, it needs to know the directory in which you have placed the CodeCheck rule files supplied by Abraxas. CodeCheck uses the `CCRULES` logical name for this purpose. Define this logical name to the appropriate directory path. For example, if your rule files are located in `DISK1:[ABRAXAS.RULES]`, then this VMS command will define the `CCRULES` logical name:

```
$ DEFINE CCRULES DISK1:[ABRAXAS.RULES]
```

If your rule files are located in more than one directory, list all directory paths in this variable, separated by commas.

If this logical name is not defined, CodeCheck will only look in the current directory for rule files. *There is no default path for rule files.*

How CodeCheck finds header files

If the filename in an `#include` directive is in double quotes, then the list of directories to be searched for header files begins with the current directory, followed by each path specified in `-Ipathname` command-line options, followed by each path specified in the logical name `C$INCLUDE`.

If the filename in the `#include` directive is in angle brackets, then the list of directories to be searched for header files begins with each path specified in `-Ipathname` command-line options, followed by each path specified in the logical name `VAXC$INCLUDE` (or `SYS$LIBRARY` if `VAXC$INCLUDE` is empty or not defined).

If the filename in the `#include` directive appears to be a full pathname, then the search list is ignored entirely.

If CodeCheck fails to find a header file then it will report a chronological list of all directories in which it looked.

Predefined macros

CodeCheck always predefines certain macros before it reads a single line of source code. Which macros are predefined, and which values they have, depend on which operating system is in use and which **–k** option is in effect. To determine exactly which macros are predefined, and their values, use the **–d?** command-line option. For example, the command `check -k4 -d?` will cause CodeCheck to print the list of macros that are predefined when generic C++ source files are checked.

In addition to the standard ANSI predefined macro constants, the following macros are predefined when no **–K** option is specified.

Vax	vaxc	vms	vax11c	CC$gfloat
VAX	VAXC	VMS	VAX11C	CC$parallel

If necessary, any predefined macro may be undefined with the **–u** command-line option, or given a different value with the **–d** option.

#5 CodeCheck Mac

Written by: Patrick Conley
Last Revised: 1 January 1997

This Technical Note discusses the behavior of CodeCheck on Macintosh computers.

CodeCheck Mac runs under MPW

CodeCheck for the Macintosh is distributed as an MPW tool, not a stand-alone application. Thus MPW (the Macintosh Programmer's Workshop) is required. Despite this restriction, CodeCheck Mac can be used by Symantec Think C users; for details see the last section in this note.

CodeCheck was originally written and developed on the Macintosh within the MPW environment. We are serious Mac developers.

How CodeCheck finds header files

If the filename in an `#include` directive is in double quotes, then the list of directories to be searched for header files begins with the current directory (the directory where the file issuing #include directives is located), followed by each path specified in `-Ipathname` command-line options, followed by each path specified in the MPW environmental variables `CIncludes` and `CPIncludes`.

If the filename in the `#include` directive is in angle brackets, then the list of directories to be searched for header files begins with each path specified in `-Ipathname` command-line options, followed by each path specified in the MPW environmental variables `CIncludes` and `CPIncludes`.

If the filename in the `#include` directive appears to be a full pathname, then the search list is ignored entirely.

If CodeCheck fails to find a header file then it will report a chronological list of all directories in which it looked.

Set the `CCRules` environmental variable

Before CodeCheck can successfully read your C source files, it needs to know the directory in which you have placed the CodeCheck rule files supplied by Abraxas. CodeCheck uses the `CCRules` environmental variable for this purpose. Set this variable to the appropriate directory path. For example, if your rule files are located in the directory `{MPW}CodeCheck:Rules`, then these two MPW commands will set the `CCRules` variable:

```
set CCRules "{MPW}CodeCheck:Rules"
export CCRules
```

If your rule files are located in more than one directory, list all directory paths in this variable, separated by commas. If this environmental variable is not set, CodeCheck will look only in the current directory for rule files.

CodeCheck recognizes the keywords `comp` and `extended`

Both Apple's MPW C and Symantec's Think C have two keywords that are not part of the ANSI standard. These keywords are `comp`, an extended integer type, and `extended`, a long double type. CodeCheck recognizes these new keywords. When a variable of base type `comp` is found by CodeCheck, the variable `dcl_base` is set to `COMP_TYPE` (this manifest constant

is defined in the CodeCheck header file `check.cch`). Similarly, when a variable of base type `extended` is found, `dcl_base` is set to `EXTENDED_TYPE`.

Note: `COMP_TYPE` and `EXTENDED_TYPE` are synonyms for the non-standard CodeCheck base types `EXTRA_INT_TYPE` and `LONG_DOUBLE_TYPE`, respectively.

How to use wildcards

CodeCheck accepts the MPW standard wildcard symbol ≈ in the name of the source file to be checked. For example:

```
check  -Rerror  ≈.c
```

will cause CodeCheck to apply the rules in `error.cc` to every file in the current directory that has the .c extension. These files will be treated by CodeCheck as though they constitute a project (*i.e.* as though they are all to be linked together after compilation). To inform CodeCheck that they are independent and not to be linked, include the **–Z** option, as in the following example:

```
check  -Rerror  -Z  ≈.c
```

Unable to swap in the Shell segment?

Some users with very large projects may get the MPW message "Unable to swap in Shell segment" while running CodeCheck. If this happens, quit MPW and increase the partition size granted to MPW by the Finder. To find the partition size, select the MPW Shell icon and pick the "Get Info" item from the File menu. The partition size is given in a box in the lower right-hand corner of the Info window.

Predefined macros

CodeCheck always predefines certain macros before it reads a single line of source code. Which macros are predefined, and which values they have, depend on which operating system is in use and which **–k** option is in effect. To determine exactly which macros are predefined, and their values, use the "**–d?**" command-line option. For example, the command `check -k4 "-d?"` will cause CodeCheck to print the list of macros that are predefined when MPW C++ source files are checked (note: the quote marks are necessary).

In addition to the standard ANSI predefined macro constants, the following macros are predefined when no **–k** option is specified:

mc68000	**MC68000**	**m68k**	**macintosh**	**applec**

If necessary, any predefined macro may be undefined with the **`-U`** command-line option, or given a different value with the **–D** option.

Symantec C++ 7.0x is supported by CodeCheck Mac

Programmers who prefer the environment offered by Symantec C++ will need to gain some familiarity with Apple Computer's MPW (Macintosh Programmer's Workshop) if they want to use CodeCheck. MPW is a Unix-like command-line environment for C programmers, and CodeCheck is an MPW tool designed for a command-line environment.

CodeCheck needs to know which folders contain header files before it can read your source files. Unlike Symantec C++, neither MPW C nor CodeCheck will search subdirectories while looking for headers. Each subdirectory containing header files must be explicitly named! This is a real nuisance if you are using the Think Class Library, so we suggest the following strategy: make a new folder called SCIncludes within the Interfaces folder of the MPW folder (***not*** within the Symantec C++ folder). Copy *every* Symantec C and C++ header into this new folder. This results in a single folder containing every Symantec header. Now you can set the `CIncludes` environmental variable to point to this one single folder, as shown below:

```
set CIncludes "{MPW}Interfaces:SCIncludes"
export CIncludes
```

The second step is to create a copy of the Symantec header file "`Mac #includes.cpp`" for use with CodeCheck. First duplicate this file, then move it to the SCIncludes folder. Do the same for the file "`TCL #includes.cpp`" if you are using the Think Class Library. Lastly, place the following conditional code at the start of every source file:

```
#ifdef CODECHECK
#include <Mac #includes.cpp>
#include <TCL #includes.cpp>   // only for TCL users
#endif
```

When using Symantec Think C, with or without its object extensions, be sure define the macro `THINK_C` on the command-line, to ensure compatibility with Think source code:

```
check -k4 -g -dTHINK_C=6 -rError mycode.c
```

Metro-Werks Code Warrior is supported by CodeCheck Mac

CodeCheck is an MPW tool, however CodeWarrior does support tools within its environment. CodeCheck will parse CodeWarrior C++ source code. Code Warrior tools may be ran from MPW thereby creating complete compatibility with CodeCheck. CodeCheck supports both the Macintosh and Microsoft Windows version of Metro-Werks CodeWarrior.

#6 CodeCheck Windows 2000 [NT]

Written by: Patrick Conley
Last Revised: 15 January 2002

This Technical Note provides basic information for using CodeCheck with Microsoft NT.

The executable file on distribute disk is named chknt.exe. This is a native WIN32 version, it is fully compatible with Windows 95 and Windows NT Workstation and Server. This version supports long file names.

To run CHKNT, you must have a 300 MHZ Pentium II (or later) computer with operating systems Windows 95 or Windows NT. CHKNT needs to run under MSDOS box invoked in Windows 95 or Windows NT. It is recommended at least 128 MegaBytes of available extended memory. And if you are running CHKNT on Windows NT, it is better to format the hard disk for memory caching to NTFS file system to improve the performance.

Different from file systems of MSDOS and Windows 3.x, Windows NT and Windows 95 allow the name of a file or directory to be formed by multiple words which are separated by blank spaces. When specifying any file or directory in command line, please put the entire path in a pair of double quotes as whole, e.g. **-I"Dir 1\Dir2\Dir 3"**. Do not specify it as -I"Dir 1"\Dir2\"Dir 3".

The DOS version of CodeCheck [chk32.exe] will operate on Windows NT/95 using as a DMPI process. However this version will not support long file names or directory names with embedded space.

CodeCheck / NT-2000 Performance

CodeCheck / NT is happiest with 256 Megabytes of RAM. If you intend to analyze very large Windows projects we highly suggest 512 Megabytes DDR to avoid page faulting. A 2 Ghz Pentium-4 should be considered minimum for interactive CodeCheck analysis on Microsoft MSDEV 7.0 or higher C++. The default paging disk must be formatted as NTFS.

Most users of CodeCheck / 2000[NT] are analysing Microsoft Visual C++ 7.0 or later source code. The *Windows.h* header file used by Microsoft Developer [MSDEV] C++ is highly recursive. Microsoft uses pre-compiled header files to solve their performance problems. CodeCheck must read each header file at least once for every module to correctly build internal symbol tables. In summary analyzing MSDEV C++ with CodeCheck requires a minimum of 256 MB DDR for large Windows C++ projects.

CodeCheck / NT can be installed as an MSDEV tool and be directly invoked from within the MSDEV enviroment and is fully compatible with the Visual Editor supplied by Microsoft. See Chapter #23 in this document

#7 Troubleshooting Syntax "Errors"

Written by: Patrick Conley
Last Revised: 15 December 1997

This Technical Note provides strategies for overcoming syntax "errors" found by CodeCheck.

Why syntax errors occur

It sometimes happens that code which compiles without error on your C or C++ compiler will generate a syntax warning or fatal error when scanned by CodeCheck. Experience has shown that the possible causes for these errors are, in decreasing likelihood, as follows:

- A macro whose definition is required by your system or library header files was not defined in the command-line. (for example)

```
check -D_WIN32 test.c // Microsoft VC++ 5.0 may require explicit OS.
```

- You specified the wrong **–K** option, so CodeCheck failed to recognize a special keyword or macro. (The default is ANSI-C otherwise explicit setting is required)

```
check -K4 test.cpp // When checking AT&T C++ -K4 must be explicit.
```

- Your compiler has one or more nonstandard keywords that are not known to CodeCheck.

```
check -d__handle="" test.c // CodeCheck treats keyword as NULL.
```

- There is a bug in CodeCheck that we need to know about.

The first and most important step

First run CodeCheck again on the same source file, but use the command-line options –**H,** –**M**, and **–D?**, and do ***not*** use –**J**. This will create a listing file named check.lst with all headers listed, all macros expanded, and all syntax error messages shown in context. Open this listing file and examine the *first* syntax error. This is necessary because later syntax errors could be propagated results of this first syntax error, by removing (or commenting out) the first syntax error, the following warnings or fatal error messages could be solved. In addition, the **–D?** option will cause CodeCheck to print a list of all macros that were predefined by CodeCheck for this run.

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

Nonstandard keywords

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

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

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

This rule will cause the CodeCheck lexical analyzer to skip over every occurrence of `packed`, `unpacked`, and `interrupt`. Try checking your code again with a rule like this. If it now parses without error then you have found a solution. As another example, this rule will prevent syntax errors for users writing for the old Zortech C compiler:

```
if ( mod_begin )
      {
      ignore( "__handle" );
      undefine( "_MSC_VER" );
      define( "__ZTC__", "0x0310" );
      define( "asm", "_intrinsic_" );
      }
```

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

```
check -K0 –Dvoid=int foo.c
```

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

If types such as **size_t** are at the place of a syntax error, check if the type is a built-in type for your compiler. If so, you can overcome the error by using a macro defined with the -D option on the command line like **`-Dsize_t=int`**.

Other types in this category are **size_t, time_t, clock_t, fpos_t, div_t, idiv_t**. Some possible reasons for are the following:

a.) The header files defining these types are not defined before they are referred in your code. These types are defined in **stdlib.h, time.h**, and **wchar.h**, etc., for most C++ compilers. Please check your project source code to see if these header files are being processed. You can do this by specifying **-P**, **-L** and **-H** on the CodeCheck command line and inspect the file **CHECK.LST** to see if these header files were processed and the types were defined.

b.) The code defining these type are included with the header files. However the definitions are suppressed by conditional compilation directives. You can see if this is the case by checking if the lines with the definitions are not macro expanded (**-M**), if this is the case then use **-D** or **-U** to expand the definition of the type. The best solution to this problem is go back to your standard compiler and do a pre-processor expansion (typically -E option) and follow the conditional path taken, where you notice a divergence with CodeCheck's (**CHECK.LST**) expansion will be the source of the problem.

c.) Header files are pre-compiled and the original compiler header files are not in the directories. In this case, the only solution is find the original compiler header files and place them on your system or access them from another system via your local area network. CodeCheck doesn't support pre-compiled header files and must have access to the original source version of the appropriate .H or .HPP files.

Creating new intrinsic type specifiers with `new_type`

Some compilers have nonstandard intrinsic types that are not defined in any header file. The function `new_type()` informs CodeCheck of the existence of such a built-in nonstandard type. The first argument for `new_type()` should be the new keyword itself, in quotes. The second argument should be any of the possible values of `dcl_base` *except* `DEFINED_TYPE`. (These values are defined as constants in the standard CodeCheck header check.cch). If the value is one of these:

```
EXTRA_INT_TYPE     EXTRA_UINT_TYPE     EXTRA_FLOAT_TYPE     EXTRA_PTR_TYPE
```

then CodeCheck will treat the new keyword as a new base type that is not equivalent to any of the standard types. If it is any other value then the keyword will be considered a synonym for the specified type. Here is the complete list of 25 base types as defined in check.cch:

```
#define VOID_TYPE             1
#define BOOL_TYPE             2
#define CHAR_TYPE             3
#define SHORT_TYPE            4
#define WCHAR_TYPE            5
#define INT_TYPE              6
#define LONG_TYPE             7
#define LONG_LONG_TYPE        8      // mainframe long long type
#define EXTRA_INT_TYPE        9      // nonstandard integer
#define UCHAR_TYPE            10    // unsigned char
#define USHORT_TYPE           11    // unsigned short
#define UINT_TYPE             12    // unsigned int
#define ULONG_TYPE            13    // unsigned long
#define EXTRA_UINT_TYPE       14    // nonstandard unsigned integer
#define FLOAT_TYPE            15
#define SHORT_DOUBLE_TYPE     16    // Symantec and others
#define DOUBLE_TYPE           17
#define LONG_DOUBLE_TYPE      18
#define INT8_TYPE             19    // __int8, __int16, __int32, and __int64
#define INT16_TYPE            20    // are types of Microsoft C++, Borland C++
#define INT32_TYPE            21    // and IBM VisualAge C++.
#define INT64_TYPE            22
#define EXTRA_FLOAT_TYPE      23    // non-standard float
#define ENUM_TYPE             24
#define UNION_TYPE            25
#define STRUCT_TYPE           26
#define CLASS_TYPE            27    // C++ only
#define DEFINED_TYPE          28    // any typedef name
#define EXTRA_PTR_TYPE        29    // nonstandard pointer
#define CONSTRUCTOR_TYPE      30    // C++ only
#define DESTRUCTOR_TYPE       31    // C++ only
```

Up to 64 new intrinsic types can be defined with calls to `new_type()`. Please note: use this mechanism only for type keywords that are built into your compiler, *never for types that are defined in header files.*

An Example of the use of `new_type()`

Let us suppose that your compiler has a nonstandard type specifier `int64`, which stands for a 64-bit integer type *that is not defined in any header*. This rule could be inserted into every CodeCheck rule file to handle this new keyword:

```
if ( prj_begin )
      new_type( "int64", EXTRA_INT_TYPE );
```

This rule introduces `int64` as a new integer type, not equivalent to any other integer type. Whenever CodeCheck finds a declaration with base type `int64`, it will set the variable `dcl_base` to `EXTRA_INT_TYPE`.

On Macintosh systems, CodeCheck understands the base types `extended` and `comp` to correspond to `LONG_DOUBLE_TYPE` and `EXTRA_INT_TYPE`, respectively. These Macintosh keywords do not have to be defined by the CodeCheck user. As a further convenience to Macintosh users, the special manifest constants `EXTENDED_TYPE` and `COMP_TYPE` may be used for these base types.

The `EXTRA_PTR_TYPE` is reserved for special atomic pointer types (pointer types that may not be dereferenced, *e.g.* Microsoft's `_segment` type). Microsoft C users do not need to inform CodeCheck about the `_segment` type: it is already understood by CodeCheck. *Borland C users please note:* contrary to statements in Borland's documentation, the Microsoft `_segment` keyword is ***not*** syntactically the same as the Borland `_seg` keyword. The former is an atomic base type, while the latter is an ordinary type modifier for pointers.

SYSTEM ERRORS

In case you run into system errors such as 'bus error', segmentation violations:

1.) If any rule file is used, remove the corresponding .cco file, and try running CodeCheck again.

2.) If the problem still exists, run CodeCheck without the Rule File. If problem is solved examine the Rule File for possible problems and send a copy to Abraxas via EMAIL (support@abxsoft.com).

3.) If the problem still exists, reduce the problem to a simple C or C++ example and send it to Abraxas via Email.

System Errors are extremely rare and Abraxas Software needs to known immediately if you discover one. Thank you.

Contacting Abraxas Software for Support

If you need to communicate with us the fastest way is via Email or Fax. Our Email is **support@abxsoft.com**, and the **Fax** number is USA:**503.232.0543**. In your Email or fax, please describe you problem. When contacting us always provide your valid Inter-Net reply address and your physical address so we can mail you a free update. If it is a syntax warning or a fatal error, please try the suggestions found in Tech Note #6 before sending your problem to us. It frequently helps to show us the relevant portion of a listing file, so that we can see the error message in its context. Make this listing file by running CodeCheck with the -H, -M, and -D? options. Do not use -J. The list file created by CodeCheck will have the name check.lst. Lastly, please give the information such as CodeCheck version (type check at the command line for your version), your Abraxas customer service number (Part Number or Serial Number), operating system, platform, C/C++ compiler, and your phone, fax, Email information. Finally always provide your current name and address so we can send you the current software free of charge.

#8 Errata in the CodeCheck Reference Manuals

Written by: Patrick Conley
Last Revised: 15 December, 1997

This Technical Note corrects all known errors and clarifies all known ambiguities in ***The CodeCheck Reference Manual***, dated January 1997.

The errata reported here are relevant to these options, functions, and variables:

See the Document 'master.txt' at ftp://ftp.abxsoft.com/codchk_doc/master.txt for the latest complete reference on CodeCheck functions, triggers, and variables.

#9 New Variables, Functions, Operators and Error Messages

Written by: Patrick Conley
Last Revised: 10 March 98, for version 7.51

This Technical Note describes all new CodeCheck variables, functions, operators, and messages that have been added since ***The CodeCheck Reference Manual*** went to press in October of 1995.

New Variables

`idn_constant` // Set to 1 when an identifier is an enum constant.

New Functions

*FILE * fopen(char*, int);*

*int fclose(FILE *);*

*int fread(FP, char *, int);*

*int fwrite(FP, char *, int);*

void remove_path(void); // Remove earliest include path specified by set_str_option('I',"...").

New Operators

(none)

New Warning Messages

C0050 **There is no class to inherit from.**

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

C0051 **Template <name> has not yet been declared.**

A C++ template name has been used without a forward declaration. A few C++ compilers consider this to be legal, but it is very poor programming style.

New Fatal Error Messages

E0057 **Allowed in C++ but not in C.**

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

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

One of the arguments to `strcmp` was `NULL`.

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

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

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

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

#10 Checking Microsoft C/C++ Sources

Written by: Patrick Conley
Last Revised: 15 October 2004

This Technical Note discusses how to use CodeCheck on C and C++ source code that was written for any of the Microsoft compilers. Currently CodeCheck Support MSDEV from VC 1.5 to MSDEV DOT-NET 7.0. Microsoft Visual Studio DOT-NET requires special CCP files for configuration.

Which version of Microsoft C/C++?

There are four versions of Microsoft C/C++ in use as of this date. CodeCheck needs to know which version you are using. The macro `_MSC_VER` has been predefined to the version number (*100) in every version of Microsoft C since 6.00. Here are the possibilities:

Version	OS	_MSC_VER	Notes
6.00	DOS or OS/2	600	*1990, C only*
7.00	DOS	700	*1991, C and C++*
8.00	Windows NT only	800	*1993, C and C++*
Visual C++	Windows NT & 3.1	800	*Same as v8.00*
Visual C++ 4.0	*Windows NT & 95*	*1010*	*MSDEV - v10.10*
Visual C++ 5.0	*Windows NT & 95*	*1100*	*MSDEV – v11.00*
Visual C++ 6.0	*Windows NT & 95*	*1200*	*MSDEV – v12.00*
Visual C++ 7.0	*Windows 2000*	*1310*	*Microsoft Visual C++ .NET – Version 13.10*

CodeCheck, when invoked with the –k7 command-line option for Microsoft C++, predefines the `_MSC_VER` macro as 1200. If this value is not correct for your application, and if your code depends on this version number, then you may redefine `_MSC_VER` on the command-line. Here is an example for version 7.00 of Microsoft C++:

```
check  -k7  -d_MSC_VER=700  myproject
```

New MSDEV C++ TYPES - __int8, __int16, __int32, __int64, and bool

From Version 4.2, MicroSoft Visual C++ introduced new integral **types __int8, __in16, __int32, __int64** and type **bool**. CodeCheck has incorporated them as built-in types. Normally, you do not need to pay any attention to these types. However, in you code, there is code which derived these types from other types with typedef. You need to rename these types to other identifiers by command option -D to avoid syntax error. For example, if somewhere in you code, there is declaration like "typedef unsigned bool;", you need to use command option -Dbool=MYBOOL. The macros defining the value of dcl_base are in file "check.cch".

Specify the target API with a command-line macro

The Microsoft C++ compiler is used to compile source code for a variety of different target API's. Depending on the target API for your source code, you may need to define certain macros on the command line for CodeCheck, so that the Microsoft headers will be correctly parsed by CodeCheck.

Target API	Define these macros
Windows 3.x	_WINDOWS

```
Win32 or Win32s    /MT /MD …
DLL                        _DLL
DLL with MFC               _AFXDLL
DLL protected-mode         _WINDLL

Mac                        _MAC          // -u_WIN32
```

For example, to check a Microsoft C++ source file named "foobar.cpp" that is designed to be compiled for the Win32 API, use:

```
check  foobar.cpp  -k7
```

If you use Microsoft C (but not C++) then read this!

To use CodeCheck on Microsoft C (*not* C++) source code, you must define `_MSC_VER` to the appropriate value and *undefine* `__BORLANDC__`. Here is an example for MS-C 6.00:

```
check  -d_MSC_VER=600  -u__BORLANDC__  myproject
```

Alternatively, you can insert the following rule into every rule file that you use:

```
if ( mod_begin )
      {
      define( "_MSC_VER", "600" );
      undefine( "__BORLANDC__" );
      }
```

These steps are necessary when you are checking C source code, but *not* C++ source code.

CodeCheck can be incorporated into the Visual C++ Environment

The following information is for those still using the old Microsoft Visual C++ (**MSVC**) 4.2 or earlier environment, if you using the new Microsoft Visual Developer (**MSDEV**) then see that chapter #23 of this technical note entitled MSDEV Studio.

It is extremely easy to incorporate CodeCheck into your Visual C++ environment. Simply use the Options menu to create a new entry for CodeCheck in the Tools menu. There are fields for the command line, optional arguments, the starting directory, and the menu text itself. When you create a tool entry in this way, it will appear in the Tools menu. Warnings from CodeCheck will be automatically captured in the Message window, and the F4 key will display each message together with the appropriate line in the source file, exactly as if these warnings had been generated by the Microsoft compiler itself.

Abraxas Software has also developed a standalone Windows NT/95 version of CodeCheck called CCWIN or CodeCheck Windows. If you would like a copy for evaluation send us email and ask for a copy, if your a registered NT/95 user.

Known bugs in Microsoft C++ 6.0 headers [There are NO known problems in MSDEV 7.0 "Dot-Net"]

The header files supplied with Microsoft C++ version 6.0 have a number of bugs that cause CodeCheck to issue syntax warnings. Here is a list of the bugs we have found to date:

File	Line	Problem
atlbase.h	5135	`T * p = &m_pBase[nElement]` *should have a semicolon on end of line. ;*

Atlwin.h *89* *after* class Cwindow*; two forward references should be added* -
class CNullTraits*;*
class CControlWinTraits*;*

Know Bugs When Running ATL.CPP Sample [MSDEV 6.0 C++]

Atlcom.h	3986	Forward references needed for *_ATL_EVENT_ENTRY*
	4326	Forward references needed for CcomEnum
Atlhost.h	1593	Undeclared Templates, Classes – need fwd reference

Problems in atlcom.h & atlhost.h require the following additions to beginning of the files respectively.

```
#ifdef CODECHECK  // fwd references added to beginning of atlcom.h
template <> struct _ATL_EVENT_ENTRY ;
template <> class CComEnum ;
template <> class CComEnumOnSTL;
class CControlWinTraits ;
#endif
```

Add the following to the beginning of atlhost.h:

```
#ifdef CODECHECK  // fwd reference  added to beginning of atlhost.h
class _ClassFactoryCreatorClass ;
class _CreatorClass ;
class  CAxHostWindow ;
class  CDLLRegObject ;
#endif
```

Known Bugs in RPC.H [MSDEV 6.0 C++]

File	Line	Problem
rpc.h	1	HWND is missing a forward declaration, add "struct HWND {} ;"

The forward declaration for HWND is missing from rpc.h, and must be placed at the beginning of file.

```
#ifdef CODECHECK  // codecheck conditional
struct HWND{}; // required by RPC.H at the beginning of file.
#endif
```

Microsoft Visual C++ .NET

The Microsoft Visual C++ .NET requires a special CCP file [CodeCheck Project]. Generally for testing and validating CodeCheck from the MS Visual Studio its best to first start from the command line. MS-Visual .NET requires the batch file "vsvars32.bat" to be executed to correctly define the SET INCLUDE environmental paths.

The use of the CCP file is

Chknt vc7.ccp yourfile.cpp

The following is the contents of the vc7.ccp file.

```
# Microsoft Visual C++ .NET 7 Configuration File for CodeCheck

-k7

#define

-D_WIN32

-D_DLL_CPPLIB

-D_MSC_VER=1300

#include search path's [ Its best to execute "vsvars32.bat" to config INCLUDE ]

-Id:\vc7\include

-Id:\vc7\SDK

To indicate emulation of 64 bit compiler add -D_INTEGRAL_MAX_BITS=64
```

#11 Checking Borland C/C++ Sources

Written by: Patrick Conley
Last Revised: 15 October 2004

This Technical Note discusses how to use CodeCheck on C and C++ source code that was written for any of the Borland compilers. We support all Borland C & C++ compilers for all time that means Turbo C to the new CodeBuilder C++. Generally CodeBuilder requires advanced CCP files, just email us and ask us to send you the correct CCP file for your Borland compiler.

Specify the target API with command-line macros

The Borland C and C++ compilers are used to compile source code for a variety of different target operating systems and API's. Depending on the target API for your source code, you will need to define (and possibly undefine) certain macros on the command line for CodeCheck, so that the Borland headers will be correctly parsed by CodeCheck.

Target API	Define these Macros	Undefine these macros
Win-16bit	*(none needed)*	
Win-32bit	`__FLAT__, __WIN32__`	`__LARGE__`
DLL-16bit	`__DLL__`	
DLL-32bit	`__DLL__`	
DOS-16bit	*(none needed)*	`_Windows`
DOS-32bit	`__FLAT__, __CONSOLE__`	`_Windows`
OS/2-2.1	*(none needed)*	

For example, to check a C++ source file named "foobar.cpp" that is designed to be compiled for the Win32 API, use:

```
check  foobar.cpp  -k6  -d__FLAT__  -d__WIN32__  -u__LARGE__
```

Known bugs in Borland's C++ 4.0 headers

The header files supplied with Borland C++ version 4.0 have a number of bugs that cause CodeCheck to issue syntax warnings. Here is a list of the bugs we have found to date:

File	Line	Problem
iomanip.h	69	`IMANIP(typ)` *should be* `imanip<typ>`
binimp.h	206	template class definition should end with a semicolon
binimp.h	441	`IteratorOrder` *should be* `TBinarySearchTreeBase::IteratorOrder`
arrays.h	577	template class definition should end with a semicolon
bags.h	454	template class definition should end with a semicolon
binimp.h	206	template class definition should end with a semicolon
deques.h	1076	template class definition should end with a semicolon
queues.h	521	template class definition should end with a semicolon
stacks.h	753	template class definition should end with a semicolon

Some Borland header files fail to provide several forward declarations for template classes that are used within template definitions. These forward declarations are ***necessary*** for CodeCheck, so that it can properly parse the templates at the time they are defined. Here are the details:

1. Add these three lines to the beginning of header file `listimp.h`:

```
template <class T, class Alloc> class TMSListImp;
template <class T, class Alloc> class TMIListIteratorImp;
template <class T, class Alloc> class TMISListIteratorImp;
```

2. Add these three lines to the beginning of header file `dlistimp.h`:

```
template <class T, class Alloc> class TMDoubleListElement;
template <class T, class Alloc> class TMSDoubleListImp;
template <class T> class TSDoubleListImp;
```

3. Add this line to the beginning of header file `binimp.h`:

```
template <class T> class TIBinaryTreeInternalIterator;
```

Processing Borland CodeBuilder C++ 6.0

```
The Borland CodeBuilder 6.0 requires a CCP file [ CodeCheck Project ]. The usuage is


Chknt bc6.ccp yourfile.cpp

The following is the contents of the CCP file for CodeBuilder 6.0 always contact
Abraxas Software for the latest CCP files.

-k6

#undefine

-U_M_IX86

-U_WCHAR_T_DEFINED

-U__TURBOC__

-UMSDOS

-U__TCPLUSPLUS__

-U_WCHAR_T

-U_X86_

-U__WIN32__

-U_MSC_VER

-Ui386

-U__BORLANDC__

-U__BCPLUSPLUS__

#define

-D__TLS__=1
```

```
-D__FLAT__=1
-D_Windows=1
-D__TEMPLATES__=1
-D__CDECL__=1
-D__CONSOLE__=1
-D_MT=1
-D__CGVER__=512
-D__MT__=1
-D__BCOPT__=1
-D_CPPUNWIND=1
-D_WIN32=1
-DWIN32=1
-D__WIN32=1
-D__WIN32__=1
-D__BOOL__=1
-D__TCPLUSPLUS__=0x0560
-D__BORLANDC__=0x0560
-D__TURBOC__=0x0560
-D__BCPLUSPLUS=0x0560
-D_M_IX86=500
# conflict of using 'isdigit' as template & macro
-D__USELOCALES__
#include
-Ic:\BC6\include
```

#12

Written by: Loren Cobb
Last Revised: 17 May 1994

This Technical Note discusses how to use CodeCheck on source code that was written for the Symantec C and C++ compilers running under Windows. For notes on the Macintosh versions of these compilers, please see Tech Note #5.

Specify the target API with command-line macros

The Symantec C and C++ compilers are used to compile source code for a variety of different target operating systems and API's. Depending on the target API for your source code, you will need to define (and possibly undefine) certain macros on the command line for CodeCheck, so that the Symantec headers will be correctly parsed by CodeCheck.

Target API	Define these Macros	Undefine these macros
Win-16bit	*(none needed)*	
Win-32bit	`__NT__`	
DLL-16bit	*(none needed)*	
DLL-32bit	`__NT__`	
DOS-16bit	*(none needed)*	
DOS-32bit	`__NT__`	
OS/2	`__OS2__`	
Xenix	`M_XENIX`	
Unix	`M_UNIX`	

For example, to check a C++ source file named "foobar.cpp" that is designed to be compiled for the Win32 API, use:

```
check  foobar.cpp  -k5  -d__NT__
```

#13 Checking Watcom C/C++ Sources

Written by: Loren Cobb
Last Revised: 17 May 1994

This Technical Note discusses how to use CodeCheck on source code that was written for the Watcom C and C++ compilers.

Always define `__segment` and `__WATCOMC__`

The Watcom header file xxx.h uses the identifier __segment, which CodeCheck normally interprets as a reserved keyword. To prevent a fatal syntax error, use these options in your command line:

```
check -d__segment=__watseg  -d__WATCOMC__=800
```

Specify the target API with command-line macros

The Watcom C and C++ compilers are used to compile source code for a variety of different target operating systems and API's. Depending on the target API for your source code, you will need to define (and possibly undefine) certain macros on the command line for CodeCheck, so that the Watcom headers will be correctly parsed by CodeCheck.

Target API	Define these Macros	Undefine these macros
Win-16bit	`__WINDOWS__`	
Win-32bit	`__WINDOWS_386__`,	`__FLAT__` `__LARGE__`
DOS-16bit	*(none needed)*	
DOS-32bit	`__FLAT__`	`__LARGE__`
OS/2	*(none needed)*	
QNX	`__QNX__`	

For example, to check a Watcom C++ source file named "foobar.cpp" that is designed to be compiled for 32-bit Windows, use:

```
check  foobar.cpp  -k4  -d__FLAT__  -d_WINDOWS_386__  -u__LARGE__
```

#14 The Rogue Wave C++ Libraries

Written by: Loren Cobb
Last Revised: 22 June 1998

This Technical Note discusses how to use CodeCheck on source code that was uses the Rogue Wave class libraries.

Specify your compiler!

The Rogue Wave header files contain a large amount of conditional code that depends upon macros that are predefined by your compiler. CodeCheck will parse this conditional code correctly if you make sure that CodeCheck also has the same macros predefined. The following table shows the macros that you need to define (or undefine) depending on the target compiler:

Compiler	Define these macros	With this value
AT&T cfront 2.x	`__ATT2__`	`1`
AT&T cfront 3.0	`__ATT3__`	`1`
Glockenspiel	`__GLOCK__`	`1`
Gnu	`__GNUC__`	`1`
IBM C-Set	`__IBMCPP__`	`1 //( IBM Visual Age C++ )`
Metaware	`__HIGHC__`	`1`
Watcom	`__WATCOMC__`	`950`

For example, if you are using a Rogue Wave library in source code written for Watcom C++ version 9.5, then use the following command-line:

```
check  foobar.cpp  -k4  -d__WATCOMC__=950
```

Compilers not listed in the above table probably do not need any macros defined. In particular, CodeCheck attempts to define the proper macros for Rogue Wave for four compilers: Apple MPW C++, Borland C++, Microsoft C++, and Symantec C++. However, if you encounter mysterious syntax errors then it is possible that Rogue Wave needs to have a macro defined or un-defined. To find out, run CodeCheck again with the **–H** and **–M** options, and look at the listing file (`check.lst`) that is generated by CodeCheck. Study the part of the listing that shows the header file `compiler.h`. This is where most (but not all) of the compiler-specific conditional code is located. Look for conditional code that refers to your compiler, and see whether CodeCheck has used the correct conditions. If not, then you will be able to define or undefine the appropriate macros on the command line.

Rogue Wave and Borland C++

In the case of Borland C++, the Rogue Wave conditional code is very sensitive to the compiler version number, stated in hexadecimal. For example, in version 4.10 of Borland C++ the macro `__TURBOC__` has the value `0x410`. In you encounter difficulties then first make sure that CodeCheck is using the right version number (use the **–D?** option to see a list of macro definitions).

Rogue Wave and Metaware Ansi C

In the case of Metaware, when using there C++ compiler, but checking standard C, use the -k11 option, but undefine the default MSDOS macro (-UMSDOS). For a list of all intrinsic #define's for Metaware use **-k11** with the **-D?** option [*check -k11 -d?* <cr>].

#15 Type Checking with CodeCheck

Written by: Loren Cobb / Patrick Conley
Last Revised: 22 June 1998

This Technical Note discusses how to use CodeCheck to perform type-checking on C/C++ code.

Type-checking with CodeCheck C/C++

Beginning with version 5.04, CodeCheck performs type-checking on C source code. Type checking for C++ source code was be enabled in version 7.51. In addition, we have added 25 new variables and 12 new functions to CodeCheck that will allow users to write custom rules that detect a very large variety of type-related events.

There are four broad categories of type-related rules that CodeCheck can now enforce. Rules can now be written to detect: (a) a cast to or from any specified type, (b) implicit type conversions to or from any specified type, (c) use of functions or variables of any specified type, and (d) use of an operand of any specified type for any specified operator. In addition, CodeCheck now automatically checks function argument and return() types for compatibility with the prototype for the function, if one is in scope.

The new variables that have been added to CodeCheck fall into two new categories. Variables with the new prefix `cnv_` describe characteristics of implicit type conversions. These are the compiler-generated conversions that happen automatically when a variable of one type is assigned to a variable of another type, without an explicit type cast (for example, when a pointer is assigned to an integer). Variables with the new prefix `idn_` describe characteristics of identifiers (variable and function names), when they are used in executable code.

Three Quick Examples

1. Suppose that we want to find all occurrences of an *explicit cast* of a pointer to a `struct XYZ` to anything. The variable op_cast will act as the trigger for the rule. The cast operator has two operands: the first is type that is to be cast, the second operand is the result type. We need to detect a cast in which the first operand is a pointer to a `struct XYZ`. This type has two levels: the first is "pointer to…", and the second is the base type, namely "struct".

The CodeCheck rule will look like this:

```
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 pointer to struct XYZ" );
      }
```

The functions op_levels(), op_level(), op_base(), and op_base_name() are extremely similar to their counterparts among the dcl_ functions, except that they can be applied to each operand of any operator. Their first argument is always the operand index (1 for the first operand, *etc*.).

2. Suppose that we want to find all occurrences of an *implicit conversion* of anything to a pointer to a `struct XYZ`. Like the cast operator, there are two operands for an implicit type conversion. The first operand is the type to be converted, the second is the result type. The CodeCheck rule looks like this:

```
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, "Conversion to pointer to struct XYZ" );
      }
```

3. Suppose that we want to detect every use of any global variable in executable code. (By "global" we mean a variable with file scope and external linkage.) When these variables are found, we want to print the line number and file name where the global variable was declared. The rule will look like this:

```
if ( idn_variable )
      if ( idn_global )
            {
            printf( "Variable %16s (file %s, line %d)\n", idn_name(),
                        idn_filename(), idn_line );
            }
```

The trigger for this rule should *not* be written like this: *`if (idn_variable && idn_global)`*, because logical conjunctions in a trigger result in multiple evaluations of a rule. In this example the rule would be triggered twice: once when all rules referencing `idn_variable` are evaluated, and again when all rules referencing `idn_global` are evaluated. To avoid multiple evaluations, do not use logical conjunctions in trigger expressions.

4. Suppose we want to make sure that the string copy function `strcpy` is never called with a second argument that is longer than the first. Here is a rule that will perform the test:

```
if ( op_call )
      if ( strcmp(op_function(), "strcpy") == 0 )
            if ( (op_level(1,0) == ARRAY) && (op_level(2,0) == ARRAY) )
                  {
                  src_dim = op_array_dim( 1, 0 );
                  dest_dim = op_array_dim( 2, 0 );
                  if ( (src_dim > dest_dim) && (dest_dim > 0) )
                        warn( 1234, "strcpy destination string is too short!" );
                  }
```

The test for `(dest_dim > 0)` is necessary because `dest_dim` will have been set to –1 if the array size was unspecified in the declaration of the destination string.

New Variables for Type Checking

`cnv_any_to_bitfield`	Set to 1 when an expression requires an implicit conversion from any type to a bitfield.
`cnv_any_to_ptr`	Set to 1 when an expression requires an implicit conversion from any non-pointer type to a pointer type.

cnv_bitfield_to_any	Set to 1 when an expression requires an implicit conversion from a bitfield to any type.
cnv_const_to_any	Set to 1 when an expression requires an implicit conversion from a constant type to any non-constant type.
cnv_float_to_int	Set to 1 when an expression requires an implicit conversion from a floating-point type to an integer type.
cnv_int_to_float	Set to 1 when an expression requires an implicit conversion from an integer type to a floating-point type.
cnv_ptr_to_any	Set to 1 when an expression requires an implicit conversion from a pointer type to any non-pointer type.
cnv_ptr_to_ptr	Set to 1 when an expression requires an implicit conversion from a pointer type to a different pointer type.
cnv_signed_to_any	Set to 1 when an expression requires an implicit conversion from a signed type to any unsigned type.
cnv_truncate	Set to 1 when an expression requires an implicit conversion from a larger arithmetic type to a smaller arithmetic type.
idn_base	Set to the base type of an identifier, using the same values as **dcl_base**, *e.g.* `VOID_TYPE` or `CHAR_TYPE`.
idn_bitfield	Set to 1 if this identifier is a named bitfield.
idn_constant	Set to 1 when an identifier is an enum constant.
idn_function	Set to 1 when an identifier is a function name.
idn_global	Set to 1 when an identifier has file scope and external linkage.
idn_levels	Set to the number of levels of an identifier, using the same values as **dcl_levels**.
idn_line	Set to the line number of the declaration in scope for the identifier. For the filename of the file in which the declaration is found, use the function **idn_filename**().
idn_local	Set to 1 when an identifier has local scope (*i.e.* it was declared within a function body).
idn_member	Set to 1 when a C++ identifier has class scope (*i.e.* it was declared as a member of a class, struct, or union).
idn_no_init	Set to 1 when a local variable is used before it has been initialized.
idn_not_declared	Set to 1 when an identifier is a function name with no declaration in scope.
idn_no_prototype	Set to 1 when an identifier is a function name with no prototype in scope.
idn_parameter	Set to 1 when an identifier is a function parameter.
idn_storage_flags	Set to an integer which identifies the storage class of the identifier, using the same values as **dcl_storage_flags**.
idn_variable	Set to 1 when an identifier is a variable.

op_call	Set to 1 when a function call is about to be executed.
op_operands	Set to the number of operands expected whenever an executable operator is encountered. When the operator is a function call, this variable is set to the number of actual arguments in the argument list. When the operator is a cast, this variable is set to 2 (the first operand is the result type, the second is the type of the operand to be type-cast).
op_subscript	Set to 1 when an array subscript is about to be evaluated. This occurs *after* the subscript index has been evaluated.

Changed Variables

op_open_funargs	This is now merely a punctuation operator that is set to 1 when the open parenthesis of a function call or declaration is found. Use **op_call** if you want to trigger on the event in which the function is actually called (which occurs *after* all function arguments have been evaluated).

Obsolete Variables

op_open_subscript	Use **op_subscript** if you want to trigger on the event in which the subscript is evaluated, or **op_open_bracket** if you want to trigger on any open bracket. You can use op_declarator and op_executable to help find brackets used specifically in declarations or executable code.
op_close_subscript	Use **op_subscript** if you want to trigger on the event in which the subscript is evaluated, or **op_close_bracket** if you want to trigger on any close bracket. You can use op_declarator and op_executable to help find brackets used specifically in declarations or executable code.
exp_no_prototype	Use **idn_no_prototype** instead.
exp_not_declared	Use **idn_not_declared** instead.
stm_no_init	Use **idn_no_init** instead.

New Functions for Type Checking

```
int dcl_array_dim( int k )
```

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

```
int idn_array_dim( int k )
```

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

*char * idn_base_name(void)*

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

*char * idn_filename(void)*

Returns the name of the file that contains the declaration in scope for the identifier. For the line number of the declaration, use the variable **idn_line**.

int idn_level(int k)

Returns the kind of the k[th] level of the type of an identifier, using the same values as the function **dcl_level**(), *i.e.* `POINTER`, `ARRAY`, `FUNCTION`, or `REFERENCE`. The number of levels in the type is given by the variable **idn_levels**.

int idn_level_flags(int k)

Returns the flags for the k[th] level of an identifier, using the same values as **dcl_level_flags**(), *e.g.* `CONST_FLAG` or `FAR_FLAG`.

*char * idn_name(void)*

Returns the name of an identifier as a character string.

int op_array_dim(int j, int k)

If the k[th] level of the type of the j[th] operand of an operator is an array, then this function returns the dimension of the array (or -1 if no dimension was given).

*char * op_base_name(int j)*

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

int op_bitfield(int j)

Returns 1 if the base type of the j[th] operand of an operator is a named bitfield, otherwise zero.

int op_level(int j, int k)

Returns the kind of the k[th] level of the j[th] operand of an operator, using the same values as the funciton **dcl_level**(), *i.e.* `POINTER`, `ARRAY`, `FUNCTION`, or `REFERENCE`.

int op_level_flags(int j, int k)

Returns the flags for the k[th] level of the j[th] operand of an operator, using the same values as **dcl_level_flags**(), *e.g.* CONST_FLAG or FAR_FLAG.

int op_levels(int j)

Returns the number of levels of the j[th] operand of an operator, using the same values as the variable **dcl_levels**.

#16 CodeCheck under IBM MVS-OE

Written by: Patrick Conley
Last Revised: 13 December 2004

This Technical Note discusses the installation and execution of CodeCheck on mainframe computers that are running the Open Edition version of the IBM/MVS operating system.

Effective Winter-2004 IBM MVS C/C++ is now activated with CodeCheck –K12 switch.

Shrouded sources

CodeCheck MVS is supplied as "shrouded" C source files. Shrouding is a process that renders the source code intelligible only to C compilers. Shrouding removes all formatting and comments, encodes all identifier names, and converts high-level grammatical constructs (*e.g.* `while` and `for`) into low-level code (`if` and `goto` statements). We distribute CodeCheck in shrouded sources in order to maximize portability while still maintaining security for our intellectual property.

CodeCheck requires an ANSI compiler

The source files for CodeCheck require an ANSI-compliant compiler and preprocessor. The C89 compiler from IBM is known to compile CodeCheck successfully, but any ANSI compiler will do as well. The makefile provided with the CodeCheck sources assumes the C89 compiler.

How CodeCheck searches for header files

Important: CodeCheck cannot read headers (or any other files) that are located in "native" MVS datasets. If your C/C++ compiler header files have not already been copied into the heirarchical file system, then it will be necessary to move them there before running CodeCheck. Move them to the directory `/usr/include`, and be sure to give them their normal names (*e.g.* `stdio.h` instead of `stdio`). Be sure also to create the directory `/usr/include/sys`, and move the these headers to this subdirectory: `stat.h`, `modes.h`, and `types.h`. If your compiler also supplies these headers: `locking.h`, `timeb.h`, and `utime.h`, then move them to `sys` also.

CodeCheck MVS looks for header files in the same way as any Unix compiler. The default directory for headers is `/usr/include`. If any command-line options **–I** have been used, then directories given in these options are searched before the default directory. If desired, the **–I** command-line options can be placed in project files, one per line.

CodeCheck also looks at the `INCLUDE` environmental variable for header directories. You may set this variable to identify the header directory paths. For example, if some of your C headers are located in `/u/C89/include`, then the following C-Shell command will set the `INCLUDE` variable to the correct path:

```
INCLUDE="/u/C89/include"
export INCLUDE
```

If your headers are located in more than one directory, list all directory paths in this variable, separated by **colons**.

CodeCheck searches directories for system header files in this order:

1. Directories specified by calling the CodeCheck function `set_str_option('I', … )`.
2. Directories specified with **-I** in the command-line or in project files.
3. Directories specified in the `INCLUDE` environmental variable.

4. The default header directory, `/usr/include`.
5. The current directory.

How CodeCheck searches for rule files

CodeCheck searches for rule files in the default directory `/usr/CodeCheck/rules`, and then in the current directory. Note the two upper-case C's in the name of this default directory.

CodeCheck also looks at the `CCRULES` environmental variable for rule directories. You may set this variable to identify the rule directory paths. For example, if some of your CodeCheck rule files are located in `/usr/foobar`, then the following C-Shell command will set the `CCRULES` variable to the correct path:

```
CCRULES="/usr/foobar"
export CCRULES
```

Directories listed in `CCRULES` are searched before the default rule directory. If your rule files are located in more than one directory, list all directory paths in this variable, separated by colons.

Predefined macros

CodeCheck always predefines certain macros before it reads a single line of source code. Which macros are predefined, and with which values, depend on which operating system is in use and which **–k** option is in effect. To determine exactly which macros are predefined, and their values, use the **–D?** command-line option. For example, `check -k4 "-D?"` will cause CodeCheck to print the list of macros that are predefined when generic C++ source files are checked.

The following macros are predefined for the C89 compiler:

`Unix`	`__COMPILER_VER__`	`_Packed`
`__unix`	`__TARGET_LIB__`	`__EDC_LE`

If necessary, any predefined macro may be undefined with the **`-U`** command-line option, or given a different value with the **–D** option. The special option **–D?** will cause CodeCheck to print a list of all predefined macros.

Extremely Important: most compilers have some predefined macros. Be sure to define yours on the command-line when invoking CodeCheck. Failure to do so will result in highly mysterious syntax errors whenever a header or source file contains one of these macro names.

#17 IBM VisualAge C/C++ Compiler

Written by : Patrick Conley

Last Revised: December 26, 2004, for version 11.13B1

This Technical Note discusses how to use CodeCheck on source code that was written for IBM VisualAge C++ compiler. CodeCheck supports IBM Visual Age C++ for the OS/2 and Microsoft Windows platforms.

The IBM VA-C++ compiler is very popular on IBM-AIX operating systems.

Checking IBM Visual Age C++

From V6.05, option -k8 is changed from "Whitesmith C" to "IBM VisualAge C++".

CodeCheck, when invoked with command option -k8 for VisualAge C++, predefines macro **__IBMCPP__** to 350 which is the version number of VisualAge C++ compiler.And it also predefines macro __cplusplus. Macro _M_I386 is predefined. __va_list is defined for both cases when macros _M_I386 and __THW_PPC__ are defined in header files stdio.h and stdarg.h. If your sources intend to use __va_list defined in condition of __THW_PPC__, you can use command-line options -u_M_I386 and -d__THW_PPC__.

Specify the target API with a command-line macro

Target API	Define these macros	
Windows 3.x	__WINDOWS__	
Win32	_WIN32	// Default for Dos and NT/95 Version

Checking IBM Visual Age C++ on IBM-AIX

Normally CodeCheck issues worst case messaging for dcl_hidden. If you wish to force CodeCheck to use the latest ISO-C++ scoping rules and assume that your code will never be ported to non-standard C++ compilers then you can tell CodeCheck to ignore events that would normally trigger dcl_hidden.

```
If ( prj_begin ) {
        Set_scope_inner( 1 );       // force code-check to hide conditional decl's within the inner scope block
}
```

Below is a typical VACPP6.CCP CodeCheck Configuration for for IBM-VA C++ Version 6.0, running on IBM AIX.

```
# use -k8 to support GNU C++ and/or IBM VA
-k8
-d__OS400__
-d__except=_except_
-d__IBMCPP__=600
-D_POSIX_SOURCE
-D_XOPEN_SOURCE=500
-d__TOS_AIX__
-d_POWER
-d_power_rs
-dAIX
-d_AIX
-d_AIX43
#  header Path's required. [ AIX search path, critical MUST be this order ]
-i/usr/include
-i/usr/include/sys
-i/.
-i/usr/vacpp/include
```

#18 New Command Options & Functions

Written by: Patrick Conley

Last Revised: January 2003 Version 10.01

This Technical Note describes all new CodeCheck options and functions that have been added since CodeCheck Reference Manual went to press in January 15, 1998. For the most recent list of new trigger and/or functions please see the file **'master.txt'** found on our ftp site at ftp://ftp.abxsoft.com/codchk/master.txt.

New CodeCheck Options

-K8 changed from "Whitesmith C" to "IBM VisualAge C++"

From V6.06, option -k9 is designated for Metrowerks CodeWarrior C++.

New functions

*int all_digits(char *);*

*int all_lower(char *);*

*int all_upper(char *);*

*int atoi(char *);*

*float atof(char *);*

*void eprintf(char * format, …);* // Same as printf() except print to stderr instead of stdout.

Void pp_error_ severity(int); // Change defined PP behavior of #ERROR directive

void remove_path();

void skip_macro_ops(int); // op_ variables to be set by operators derived from macro expansion

*char *strcat(char*, char*);*

*char *strncat(char*, char*, int);*

*char *strcpy(char*, char*);*

*char *strncpy(char*, char*, int);*

int strncmp(char, char*, int);*

*char *strchr(char*, int);*

*char *strrchr(char*, int);*

int strcspn(char, char*);*

int strspn(char, char*);*

All these function have same specification as ANSI C standard library functions with same names.

void skip_nonansi_ident(char);

Since option -K8 for Whitesmith C is dropped for IBM VisualAge C++. To make CodeCheck still be able to check Whitesmith C code, this function is needed to make tell CodeCheck to skip non-standard identifiers(leaded by charaters '@', '#' and'`'). The parameter to this function is the leading character.

#19 ObjectSpace/HP Standard Template Libraries

Written by : Shuming Tan

Last Revised: December 15, 1997

Using Object Space STL with CodeCheck

When you are installing STL<ToolKit>, you will select the host C++ compiler. So when you are using ObjectSpace STL Class Libraries, you should use the header files of the host C++ compiler. What you need to do is make sure that the directories containing the header files are reachable for CodeCheck by specifying either environmental variable CCRULES or command option -I. And please use corresponding command option -K for the host C++ compiler.

When you are checking code written for cross platform development, you should specify correct symbols with option -D.

Symbol	Platform
OS_WIN_3_1	Windows 3.1, Windows for Workgroup 3.11
OS_WIN_NT_3_1	Windows NT 3.1
OS_WIN_NT_3_5	Windows NT 3.5
OS_WIN_NT_4_0	Windows NT 4.0
OS_DOS_6	MS_DOS 6.x
OS_WIN_95	Windows 95
OS_SOLARIS_2_4	Solaris 2.4
OS_SOLARIS_2_5	Solaris 2.5
OS_OSF1_4	Dec Alpha running OSF1 4.x
OS_WIN_32	All Microsoft Windows 32 Platforms
OS_OS2_3	IBM OS/2 3.x
OS_SUN_4_1	Sun OS 4.1
OS_HPUX_9	HP UNIX 9.x

OS_HPUX_10	HP UNIX 10.x
OS_IRIX_5	Irix 5.0
OS_AIX_4	IBM AIX 4.x

Debugging OBJECTSPACE/HP STL and CodeCheck

Correct operation of CodeCheck in conjunction with ObjectSpace STL requires at least CodeCheck 6.05.

If you encounter mysterious syntax errors then it is possible that STL needs to have a macro defined or undefined. To find out. run CodeCheck with the options -H and -M options, and look at the listing file(check.lst) which is generated by CodeCheck with option -L. If you have difficulty in findng out the cause. Please email the listing file to Abraxas technical support with the problem addressed. See section #6 Trouble Shooting for complete information on debugging these types of problems. Lastly, please share your problems with Abraxas Software via email at support@abxsoft.com.

#20 NameSpace - ANSI C++ Working Draft

Written by: Shuming Tan

Last Revised: January 6, 1997

This Technical Note discusses how to use CodeCheck with C++ source code that references ‘namespace’ as defined by the 1996 ANSI C++ Working Draft.

From V6.06, CodeCheck is able to parse syntax of namespace which is described in " Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++" (Doc No:X3J16/95-0087 WG21/N0687). The description about namespace also can be found in "Microsoft Visual C++ Language Reference" (published by Microsoft Press).

Namespace now is available in C++ compilers such as Microsoft Visual C++, Symantec C++, etc.

Basically, a namespace is an optionally named declaration region. The entities declared within a namespace can be referred by the name of the namespace. A namespace can be either named or unnamed. Also a namespace can be split into multiple parts at different locations. Like class, namespace can be declared in a nested way. However, namespace can not be declared in function and class definition. For example,

```
namespace A {
      int i;
      typedef char Foo;
}

namespace {             // unnamed namespace
void f() { }
}

namespace A {           // namespace splitted into multi-parts
      namespace B {     // nested namespace
class C{ };
      }
}
```

A declared namespace can be used in two ways.

1. As qualifier to a name declared within the namespace, the qualified name can be class name, type name, variable name, function name, template name or namespace name which is nested within the qualifying namespace. Like A::i++

2. Used by being declared in using directive. For Example,

```
namespace A { int i; }

using namespace A;

void f() { i++; }   // A::i++
```

An unnamed namespace behaves as if it were replaced by

namespace unique { namespace-body }

using namespace unique;

where, for each translation unit, all occurrences of unique in that translation unit are replaced by an identifier that differs from all other identifiers in entire program. For example,

```
namespace { int i }   // unique::i

void f() { i++; }   // unique::i++


namespace A {

      namesspace {

int i;      // A::unique::i

int j;      // A::unique::j

}

void g() { i++; }   // A::unique::i++;

}
```

Members of namespace can be defined either within or outside the namespace.

Currently, CodeCheck is able to accept sources with namespaces defined and used. But there is not any triggers or function specific to namespace. In the future, a new group of variables and functions carrying prefix nsp_ will be added. Your recommendations are welcome.

#21 Checking Metrowerks CodeWarrior C/C++ Sources

Written by: Shuming Tan

Last Revised: January 6, 1997

This Technical Note discusses how to use CodeCheck on source code that was written for the Metrowerks CodeWarrior C++ compilers.

Checking Metrowerks CodeWarrior C++

From V6.06, option -k9 is designated for Metrowerks CodeWarrior C++.

CodeCheck, when invoked with command option -k9 for Metrowerks CodeWarrior C++, predefines macro **__MWERKS__** and **__cplusplus**.

On Windows 95/NT, as noticed, the names of some files and directories can consist of words separated by blank spaces. When specifying this type of file or directory names in command line, please put the entire path in a pair of double quotes as a whole.

#22 Checking SUN C/C++ Code on SUN Sparc

Written by: Patrick Conley

Last Revised: 18 June 1997

This Technical Note describes caveats related to running CodeCheck on the Sun Sparc and/or Solaris Operating System.

Parsing C/C++ on the SUN SPARC

It is recommended to define macro **__sparc** or **sparc** with command line option -D. If the macros are not specified in command line, CodeCheck may encounter a fatal error with message "ISA not supported". This is because the included header file */usr/include/sys/isa_defs.h* requires a macro specifying the target machine. If the macro is undefined, the conditional compilation will go to line "#error ISA not supported" which causes the fatal error.

Solaris machine dependent caveats

Please define one of macros **__i386**, **i386**, **__ppc**, **__sparc**, or **sparc** for the machine type with command line option -D. Using CodeCheck with Solaris C/C++ requires that one of these macros be defined. It is possible to use any platform and check your source for the appropriate target by defining the correct macros.

#23 Running CodeCheck within Microsoft Visual C++ Developer Studio

Written by: Patrick Conley

Last Revised: 15 January 2003

This chapter covers integrating CodeCheck with Microsoft Visual C++ Developer Studio. For complete integration of CodeCheck and MSDEV as a GUI please see our GoCheck Software.

How to integrate CodeCheck with Microsoft Developer Studio

Microsoft Visual C++ Developer Studio provides a way to incorporate CodeCheck into its environment.

Following are the steps how to add CodeCheck into Developer Studio.

Step 1. Pull down ***Tools*** menu and choose command ***Customize***, a customize dialog box with pops up.

Step 2. In dialog box, choose tab ***Tools***, you will see the dialog for adding a tool.

Step 3. To add CodeCheck,

a). Click on ***New***, in the new field within ***Menu contents***, type in the text which you wish to appear in pull down menu.

b). In field ***Command***, type in the path name of CodeCheck executable. You also can use ***browse*** button besides the input field to set the path name.

c). In field ***Arguments***, type in the necessary command options for run a successful checking, such as -K7, -D_WIN32 etc. It is recommended that you put the option used most of time.

Step 4. You can toggle two more choices,

a). *Use* ***output*** *window*, if this choice is on, all the output from CodeCheck will go to ***output*** window of studio. Otherwise, all the output will go to a DOS window popped up for the output. It is recommended to set this choice on. With the messages in ***output*** window, by double clicking on the CodeCheck message line with file name and line number, the window containing the file will be displayed and the line of the line number in the warning message will be pointed. You also can you use key F4 to navigate through the CodeCheck messages.

b). ***Prompt for arguments***, if this choice is on, a dialog box will be popped up to remind you if you want modify command arguments. If quite often you run CodeCheck you need to specify something different in command options. It is recommended to put the fixed part in ***Command arguments*** field. Add the changing parts in this dialog box.

Step 5. Click on button ***Close***, CodeCheck has been added into ***Tools*** menu.

To run CodeCheck, simply just pull down ***Tools*** menu and choose the command for CodeCheck.

Searching for Header Files within MSDEV STUDIO

Though Developer Studio provides the way to set the directories for searching included header files, the directories set in this way have no effect on CodeCheck's searching header files. CodeCheck still looks for header files base on three sources, current working directory, the directories specified by environmental variable **INCLUDE** and directories specified in command options -I(both in command line or rule function call **set_str_option('I',...)**). If any rule file is used without full path name specified, environmental variable CCRULES also need to be set.

Checking Projects and individual files with MSDEV STUDIO

To check single source file, you can either specify the detailed file name in command argument or specify the file name as $(FileName)$(FileExt) to make CodeCheck to check the file in top window.

To check a project, you can either specify the CodeCheck project file name in command arguments or specify the CodeCheck project file as $(FileName)$(FileExt) which represents a project file open as text file shown in top window.

Having source files in different directories:

Instead of using $(FileName)$(FileExt) use $(FilePath). This will open a project whose .dsw file resides in one directory, and still be able to check a file which is part of that project but resides in a different directory.

#24 Improving CodeCheck Speed

Written by: Shuming Tan

Last Revised: 15 December 2002

This chapter covers integrating CodeCheck with Microsoft Visual C++ Developer Studio. Also relevant is Pre-Compiled headers. There are many ways to make CodeCheck very fast when parsing MicroSoft C++ please contact us for the latest techniques.

Codecheck on a relatively large Microsoft C++ project

When running Codecheck on a relatively large project, it is noticed that Codecheck will spend significant amount of time on parsing included header files. Since the inclusions of headed files occurs at source file level, some header files will need to be included for most of the source files in the project, either directly or indirectly. To speed up the process, we can form a pseudo module pmain.c that includes all the source files as header files. For examples, suppose we have a project will has 3 source files f1.c, f2.c and f3.c. And each file includes header file f4.h. If we check f1.c, f2.c and f3.c as individual source files of a project, we will see that header file f4.h will be fully included 3 times, one for each source file. It is possible that file f4.h could be included more than once in within a source file, either directly or indirectly. Command option –G can help to reduce the inclusion of file f4.h in a source file to once. However, it is still necessary to have an full inclusion for the source file. In most of the cases, the header file is included to provide the same information. Therefore, they can be fully included just once. Noticed that if the header file has a proper wrapper, the inclusion other than first one in a source file can be takes the amount of time that can be ignored. The source files f1.c, f2.c and f3.c are included as header files in pseudo module pmain.c

```
/* pmain.c */
#include "f1.c"
#include "f2.c"
#include "f3.c"
```

Header file f4.h will be included 3 times indirectly in pseudo module pmain.c. However, if the header file has a proper wrapper, except the first inclusion, the ensuing inclusion will be just scanning of lines of the header file wrapper. The large portion of the header file will be skipped. In this case, header file is actually fully included once.

Noticed that it is assumed that a header file is included in same way in all source files. Otherwise there will be the same problem raised by using command option -G. See P. 3 of CodeCheck Reference for the explanation.

If the source files to be included are scattered in different locations, you can use relative paths and command options -I to make them be included properly just like other header files.

Because these source files are included as header files, triggers mod_begin and mod_end are not applicable to them. To get the name of the source file, function file_name() instead of mod_name() should be used. When pp_include_depth has value 1, it is referring source files instead of header files. If this method is used for faster processing and the issues mentioned above are involved in the rule files to be used, it is needed to modify the rule files so that they can adapt to this scenario.

As we know, command option -S has no effect on source file level lines. However, when this method is used, the source files are included as header files and will be treated the same as header files regarding the command option -S. To make rules be able to be applied on these source files, it is necessary to specify command option -S with value 1 or 3 explicitly. Also function set_header_optS() can help decide to which header files the rules will be applied on and in which way the rules will be applied.

#25 Extending CodeCheck Functionality

Written by: Shuming Tan

Last Revised: 10 March 1998

This Tech-Note covers extending CodeCheck's built-in custom programming features.

Extending CodeCheck

CodeCheck is not the last stop in your whole process. Combined with other programs, it can provide you more powerful results. In some cases, it could be difficult to accomplish the goal only by CodeCheck. For example presented in this section, to calculate the data complexity of a project (a set of related C source files).

Assume that there are M functions defined project-wide, for each function,

```
FAN-IN(N) = the number of functions by which this function N is called.

FAN-OUT(N) = the number of function called by function N.

IO-VARS(N) = the number of global variables used by function N + the number of
returning value.

COMPLEXITY(N) = FAN-OUT(N) / ( 1 + IO-VAR(N) )
```

The overall complexity of the project is calculated with following formula.

```
COMPLEXITY = ( COMPLEXITY(1) + … + COMPLEXITY(M) ) / M
```

The functions counted in this model are only those user defined functions, this means that the functions called but undefined will be seen as library functions.

There are some difficulties to have the job done with a single rule file.

We need to maintain a symbol table to keep and retrieve certain information when a function is defined or called. Currently, the data types provided by CodeCheck are limited. With these types, it is quite difficulty to construct and manipulate a symbol table which is mostly based on types array, struct and pointer.

CodeCheck works in the way of forward chaining and it does not retain any previous information. To keep the information available for late processing, it is user's duty to

As we noticed that data types provided by CodeCheck for implementing rules is limited, especially for symbol table manipulations. C/C++ provides more powerful and sophisticated data types. Therefore we use a 2-step strategy as the solution.

1. Use CodeCheck rules to extract necessary information from the modules. To make the information available for late processing, the information should be stored into external file(s). CodeCheck provides some I/O functions and string operating functions which are the same as or similar to the corresponding ANSI library functions.

Extending Function Meaning

fopen() Open a file with specified name and mode

fclose() Close a file.

fprintf() Output to a file.

sprintf() Output to a string.

strcat(), strncat() Concatenate two strings into one.

strcpy(), strncpy() Copy a string into another.

With these functions, we can store the extracted information into external files in relatively simple and straight forms.

2. Use the files generated in step 1 as input to a user developed program which is written specifically for processing the information we extracted from source files in previous step. With this program, we will get the results desired.

The complete solution to this problem can be downloaded from the Abraxas Web site at www.abxsoft.com. ZIP location www.abxsoft.com/dl/ccrules.zip, or can be requested by email support@abxsoft.com.

#26 GNU-GCC C/C++ Configuration

Written by: Patrick Conley

Last Revised: 13December2004

This Tech-Note covers CodeCheck's ability to analyze GNU-GCC for ALL operating systems C/C++ configurations. For a completely discussion on this issue of compiling GCC on any operating system with codecheck see the article "Using Linux Standard Base with CodeCheck".

Effective 13December2004 GCC mode is now activated by the –K13 switch.

GNU-GCC Overview

All the following is done from the standard command-line-console interface. This is to ensure independence from the wide variety of GUI's that have nothing to do with GCC compilation. Once the configuration is done, and CodeCheck is compiling your gnu-gcc C/C++ on your development workstation then it's a simple matter to invoke CodeCheck from your favorite GUI. The issue of running CodeCheck from Gui's is found in separate Technical Notes. The standard UNIX paradigm for invoking products such as CodeCheck within standard UNIX Programmer Development Environment's [PDE's] is simply installing it as a "LINT" type external subsystem. This way you can select text from the GUI and have CodeCheck process the code, and click on the results which takes you to the correct line in the file.

Here are the two standard references we use, its very important to configure your gcc comiler at the beginning. We assume in this example that the goal is to emulate only the default gcc compiler. Should another compiler need to be emulated by codecheck, then the 'gcc' shown should be substituted with the proper invocation. [Note: All examples below must be done from the Shell and/or Console/Cmd-Line.]

[See several paragraphs down for exact definition of hello.c & hello.cpp]

```
gcc -c -v hello.c > hello.c.txt                 // pipe output to hello.c.txt
```

[your system may require "2>" rather than ">" to capture the stderr output.]

```
gcc -c -v hello.cpp > hello.cpp.txt             // pipe output to hello.cpp.txt
```

From the above we can build you a custom gcc_c.ccp & gcc_cpp.ccp for your system. For testing and obtaining configuration data we use a standard set of C & C++ files. No modification must be done to these files. They must applied exactly as shown. Basically the -v tells gcc to dump the explicit path's and #defines [macros] used to process the test code. Since this information is machine specific we have found this to be the most accurate way to obtain configuration information from gcc for your development workstation.

```
//HELLO.C     GCC C Case                        // MUST BE JUST LIKE THIS
#include <stdio.h>
main() { printf("hello"); }

//HELLO.CPP   GCC C++ Case                      // MUST BE JUST LIKE THIS
#include <iostream.h>
main() {  cout<<"hello"; }
```

Operating System Independent GCC Configuration

Today, users are processing GCC with CodeCheck on IBM-AIX, SUN-Sparc/Solaris, MS-Windows, and of course Linux in all flavors. There are others, but I have just listed the most common. CodeCheck supports ALL GNU-GCC for ALL operating systems. Below we discuss the GCC on MS-Windows case, the other cases are similiar. All information to construct the CCP files shown that shown below, came directly from the "cc -c - v hello.c" information found above.

The normal mode of operation for CodeCheck is just "check filename.c" for C, and "check -k4 filename.cpp" for standard AT&T C++. However for supporting the compilers on the market we have found that Codecheck Configuration Project files, hereafter called CCP files

are the best approach. When using CodeCheck all arguments that are not switches are treated as source files, other than the CCP file. You can have as many CCP files on a CodeCheck invocation as you wish. Note a CCP file uses the pound sign [#] as a comment similar to a standard makefile. One of the most important things about CCP files is that order of include path's must be given exactly as found in the test dump, .e.g. "cc -c -v example.c". The macro values shown in the verbose compilation dump from gcc must be enumerated exactly as shown in the dump.

GCC C On Windows 2003

Given the "hello.c" case above compilation with codecheck on windows console command line would be …

```
check gnu_c.ccp hello.c
```

The contents of gnu_c.ccp are ...

```
#gnu_c.ccp                          // start of file

#force CodeCheck into GCC compiler mode -k13
-k13
-D__inline__=__inline

# gcc C macros for 686 cygnus
-D__GNUC__=3
-D__GNUC_MINOR__=2
-D__GNUC_PATCHLEVEL__=0
-D__GXX_ABI_VERSION=102
-D_X86_=1
-D__NO_INLINE__
-D__STDC_HOSTED__=1
-Di386 -D__i386
-D__i386__
-D__tune_i686__
-D__tune_pentiumpro__
-D__tune_pentium2__
-D__tune_pentium3__
-D__i386__
-D__i386
-D__CYGWIN32__
-D__CYGWIN__
-Dunix
-D__unix__
-D__unix

# #include search paths for C cygnus 686
-I/usr/include/w32api/
-I/usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/
-I/usr/include/

#GNU_C.CCP                  // end of file
```

GCC C++ On Windows 2003

C++ is is very similiar to C above, basically the major change is the path's for the header files. GCC being a modern compiler operates in C++ mode by default. The macros in the above C case are largely de-activating C++.

Command line console Usage is ...

```
check gcc_cpp.ccp hello.cpp
```

```
#GCC_CPP.CCP  // START OF FILE
# config codecheck for GCC/GNU extensions
-k13
# cygnus 686 gcc c++ macros
-D__GNUC__=3
-D__GNUC_MINOR__=2
```

```
-D__GNUC_PATCHLEVEL__=0
-D__GXX_ABI_VERSION=102
-D_X86_=1
-D_X86_=1
-D__NO_INLINE__
-D__STDC_HOSTED__=1
-Di386
-D__i386
-D__i386__
-D__tune_i686__
-D__tune_pentiumpro__
-D__tune_pentium2__
-D__tune_pentium3__
-D__i386__
-D__i386
-D__CYGWIN32__
-D__CYGWIN__
-Dunix
-D__unix__
-D__unix
# cygnus c++ header file path's
-I/usr/include/w32api
-I/usr/include/c++/3.2
-I/usr/include/c++/3.2/i686-pc-cygwin
-I/usr/include/c++/3.2/backward
-I/usr/lib/gcc-lib/i686-pc-cygwin/3.2/include
-I/usr/include
```

#27 New CodeCheck Variables, Functions, Operators and Error Messages

Written by: Patrick Conley
Last Revised: 26 December 2004, for version 11.13

This Technical Note describes all new CodeCheck variables, functions, operators, and messages that have been added since ***The CodeCheck Reference Manual*** went to press in September 2004. Most of these new triggers/functions were added to support GNU/GCC compilers, and support advanced C++ style guidelines.

New Variables

dcl_exception	C++ exception declaration complete.
dcl_local_dup	Signal if a symbol is used more than once at current local scope. The Gnu-Compiler allows this declaration, but warns "shadowed variable".
dcl_throw_parameter	A C++ throw argument. Created to support java-style exception checking.
idn_exception	Actual usage of C++ exception in code.
idn_exception_base	C++ exception base type at trigger point idn_exception.
lex_long_long	Signal "long long" 64 bit type.
lex_uc_long	Signal 'L' long type constant.
lin_within_namespace	*True if current declaration is within namespace definition.*
mod unused_static	Fires at the end-of-module when each unused static is found. The mod_unused returns the total count. The idn_name(), idn_filename(), and idn_fileline() may be used to determine where the un-used static was declared.
op_call_overload	This C++ method call is overloaded. Function return type is context dependent. Function exp_base_name() may be used to determine actual return type.
op_macro_arg	A macro function call argument.
op_macro_begin	The begin point of a macro function call. End point is signaled by op_macro_call.
pp_else	Fires on the #else macro used within #ifdef … #endif pairs. Used to verify that ALL #ifdef blocks contain a valid #else case which is common in coding standards.

New Functions

advise(int)	Enable/Disable CodeCheck Internal Warning Messages from Output. *void advise (int on_off).*
exp_base_name()	Return base-name of current expression. Useful for obtaining resultant base-name of an overloaded function and/or pointer linked overloaded functions.
find_root(char)*	Find root base type of name symbol. Useful for advanced symbol table lookup algorithms. Find_root(char * symbol-name).
find_scoped_root(char,char*)*	Find root of symbol using an explicit scope name. find_scoped_root(char *scope-name, char * symbol-name).
idn_exception_name()	Name of exception currently being used at trigger point idn_exception.
idn_fileline()	The current line number of the current 'found' identifier. Used by find_root() to determine exactly where the 'found' identifier is located.
idn_filepath()	The name of the 'found' file. Used as idn_fileline() above.
namespace_name()	*The name of the current namespace [lin_within_namespace]*
next_token()	Next token in source stream from current position. Value may be NULL at end of line. To be used with prev_token(), token(), and next_token(). Typically at any event token() returns character string name of current token.
pp_if_search(int)	Enable GNU-GCC #if (types) pre-processor method. Open-System embedded compiler support. GNU-GCC allows #if test on actual types in addition to simple macro testing. Default for this feature is off.
set_scope_inner(int)	*Force inner scoping rule to emulate standard c++ compiler. B y default codecheck place's for-loop declarations in the outer-block so dcl_hidden can find the maximum conflicts, as many c++ compilers don't actually follow the specification. If you know your code will only be ported to a perfect C++ compiler then set_scope_inner(1) activated by if (prj_begin) will force local for-loop and if-else declarations into the following scope level. By default set_scope_inner(0) is set so that dcl_hidden will fire alerting to problems that exist in c++ implementations.*
warn_format(int)	Format output string for standard UNIX/Microsoft format or Emacs format. The default is standard message format **"filename(line-number): message text"**. Emacs includes column position. Only warn(N,"format",arg1,…) messages are effected. This option is to be used as warn_format(WARN_FORMAT_EMACS) when CodeCheck is executed from within the Emac's editor. The macro constant argument of WARN_FORMAT_EMACS is defined in check.cch.

New Operators

(none)

#29 Processing IBM 390-z/OS EBCDIC

Written by: Patrick Conley

Last Revised: 09 November 2004

This Tech-Note covers CodeCheck's ability to analyze IBM 390 C/C++ from ALL operating systems.

IBM OS/390 & z/OS-C/C++ Overview

IBM OS/390 C/C++ user source code may be pure EBCDIC or it may be a form know as IBM-273, and quite often the system /usr/include headers may be IBM-1047. Many user's of codecheck may not have access to an IBM Mainframe directly to use CodeCheck therefore we have added the ability to process native EBCDIC C/C++ on an IBM mainframe from Windows 2000 and/or Linux or any other NON-IBM-MAINFRAME environment. This discussion also applies to the IBM Mainframe version of CodeCheck which can be built on a native IBM pure EBCDIC machine, e.g. CodeCheck can process ALL IBM-EBCDIC from both IBM and non-IBM platforms.

This section will briefly describe the IBM C/C++ storage format's and discuss exactly how to process IBM C/C++ with CodeCheck from Windows-2000. For an example of any other operating system please contact Abraxas Software [support@abxsoft.com].

Processing IBM C and/or C++ from a MS-Windows machine would appear using our free codecheck demo as …

```
demont -E273 4.c -rmisra_ibm.cco -l -m -h –p
```

Note, the main difference between normal usage is the addition of the –E273 switch which tells codecheck to process all source as IBM-273 by default. The –Ribm.cco tells codecheck how to process IBM specific keywords from an asc ii machine, and the following switches generate diagnostics. The –RIBM.CCO would not be required on a IBM-MAINFRAME version of CodCheck as the IBM C keywords are built-in.

IBM OS/390 C EBCDIC Storage Example

The following code example 4.c is in IBM-273 [EBCDIC] format we're using our ***e273.exe*** routine to convert the sample to ASCII for display, rather than using 'cat'. This example '4.c' uses two headers <stdio.h> and <decimal.h>. If you would like any of the tools seen here just ask by contacting us by email.

e273 4.c

```
#include <stdio.h>              /* for printf() */
#include <decimal.h>            /* for datatype decimal */
#pragma margins( 1, 80 )        /* ignore all columns other than 1 to 80 */
int main(void) {
  decimal(3,0) d;
```

```
  decimal(9,2) e;
  int i;
  d = 99D;
  e = 1111.11D;
  return 0;
}
```

In the following example the system header file decimal.h is not IBM-273, but IBM-1047. The pragma tells CodeCheck to go into IBM-1047 mode. When CodeCheck returns to caller it returns to the default [-E273], unless it had a different type of encoding. This allows CodeCheck to process any IBM C/C++ from any ASCII machine. Lastly, note here we're using "***e1047.exe***" because the decimal.h is in IBM-1047 format, the –4 means just print the first four lines of the header file.

`e1047 -4 decimal.h`

```
??=ifndef __decimal
??=ifdef __COMPILER_VER__
??=pragma filetag ("IBM-1047")
??=endif
```

Complete compiling appears in the following text, note we have added the –P switch for progress to get a 'cc -v' style verbose report. You can see that CodeCheck successfully compiled the entire example and all system headers. The #pragma message "IBM EBCDIC" is added to indicate that CodeCheck is using ibm.cch to configure the IBM C keywords that are non-standard to ANSI-C. Note that we're not using any –Kn switch so CodeCheck is operating in extended ANSI-C mode.

`demont -E273 4.c -ribm.cco -l -m -h -p`

```
Abraxas Software (R) CodeCheck NT version 11.01 B9 DEMO
Copyright (c) 1988-2004, by Abraxas Software, Inc. All rights reserved.
# Rule files are in these directories:
     d:\rules\
# Reading from file "ibm.cco"
# Reading from file "ibm.cco"
Checking extended ANSI C file 4.c with rules from ibm.cc:
#  Reading header file "ibm.cch"
 IBM EBCDIC
#    Returning to file "4.c"
#  Reading header file <stdio.h>
#  Reading header file <sys/types.h>
#    Returning to file <stdio.h>
```

```
#    Returning to file "4.c"

#  Reading header file <decimal.h>

#    Returning to file "4.c"

main

File 4.c check complete.
```

IBM OS/390 C EBCDIC Summary

In the above example the IBM C code was copied directly from the IBM Mainframe and ran on Windows 2000. Normally of course the IBM system-headers would remain intact on the IBM mainframe and would be mapped to a windows 2000 drive. Such as the following example.

```
demont -E273 4.C -Rmisra_ibm.cco -L -M -H –P –IF:
```

Where "-IF:" [dash eye F colon] means to use the F: [mapped drive] for system header searching, assuming that /usr/include on the remote IBM Mainframe has been mapped to local driver "F". Certainly if the user source and header's were also 'mapped' then the analysis could be done from a non-IBM system with code at all being on the native test machine.

Using –EN where 'N' reflects the default IBM-N EBCDIC format means that ALL source must be EBCDIC. The assumption in this discussion is that the goal is to inspect IBM C/C++ from a non-IBM mainframe machine and study the code intact without modification.

The –EN switch can be used on an IBM Mainframe in pure ebcdic environment for supporting the mixed ebcdic case where the format of the user source is different than the headers.

If a user wishes to use ASCII C/C++ as user code then all the code must be ASCII as the –EN switch forces codecheck into a pure EBCDIC mode for all file input from command invocation until termination. For this reason rule-files must be compiled separately if you wish to compile them in ASCII, otherwise of course on an ASCII machine the output of the rule-files would contain EBCDIC text when ran on a ASCII machine. This is why in the above examples we're always using a .CCO file and not compiling the rule-file as a .CC file. The rule-files are pre-compiled as ASCII so the output appears correctly on the ASCII machine. Of course if you wished to generate EBCDIC output on a ASCII machine directly without using filters then you could add the rule-file compilation switch to the EBCDIC invocation and generate pure ebcdic data.

#30 New CodeCheck Triggers and Functions Version 12.50

Written by: Patrick Conley

Last Revised: 09 November 2005

This Tech-Note covers CodeCheck's new additions for 2005.

New CodeCheck 12.5 Functions:

file_timestamp()	Timestamp in char* format [Ansi ctime()] of current file_name().
find_keyword(char)*	Determine whether a given string is a keyword in C/C++. Useful for implementing rules that say you cannot use a upper case version of a keyword. Thus by lowering the case of a string you can check if it's a keyword.
*find_similar(char *)*	See if the string provided has a match to other declarations in the code database. The default is the first 8 characters, the length can be modified with set_ambig_len(). Useful for rules that say that the first N characters of all variables must be different. Returns N, where is the number of a match found of leading characters. While dcl_similiar will fire if first set_ambig_len() is the same, this function returns the actual number of leading similar characters for any given string.
*getenv(char *)*	Standard ANSI function. An example use would be to determine CPU_TYPE so that rule-files can selectively analyze source automatically on test hardware.
*idn_get_cookie(char *)*	Get cookie value associated with idn_variable or idn_function 'found' identifier. Useful for deep analysis. For instance a cookie can keep the state of all variables assigned with malloc() to determine whether free() was called.
*idn_set_cookie(char *, int)*	Set cookie allows a state to be associated with a declaration. See idn_get_cookie() for explanation of use.
set_ambig_len(int)	Set the ambiguity length for similiarity of declarations. See dcl_similiar. Default is 8. Also effects the depth of analysis of the find_similiar() keyword. Making this a large number could bog-down a machine cpu cycles.
set_expir (int, int, int)	Set expiration date of a source file for checking. This function tell's CodeCheck to disable rule-file checking for all source files that are older than the date given. The default is ALL files are checked. Order of arguments is day, month, year. The day of month as DD [1->NN]. The month of year as MM [0->NN], and the year as YYYY.

set_inner_scope(int) — Set inner scope tells codecheck to move the declaration of a for-loop into the inner scope of the loop. The default is the outside of the loop. For example *"for(int i=0; i<0; i++) {…}"* in this case when the argument is non-zero the declaration loop-counter "i" [eye] would be moved into the inner set of braces. This is a ANSI-C++ issue. Some compiler's default to this and some do not. The use of this function is to gain perfect emulation of a particular compiler. See CodeCheck trigger's dcl_hidden and dcl_local_dup.

New CodeCheck 12.5 Triggers:

op_safe_cast — Fires when the C++ <static_cast> type case is seen. See op_cast for detecting old-style "(int)Rval" style casting. If cast is done with <…> then op_safe_cast fires, if case is done old-style then op_cast fires.

idn_init_locptr — Fires when a local variable is initialized to an address in memory. This indicates that a pointer was initialized correctly. Some coding standards require that all pointers be initialized to a legal address.

idn_init_new — Fires when a variable is initialized with C++ keyword new.

idn_no_delete — Fires when a variable created with C++ keyword new did not contain a matching delete at function/method end.

idn_specifier_flags — Similar to dcl_specifiers_flags, used to obtain specifier flags at use rather than declaration. For example if a variable is being set to a negative value, but was not declared as signed, then you could detect the violation.

Trouble Report Form

Fax to: **Abraxas Technical Support**
Fax number: **503-232-0543**
Email: **support@abxsoft.com**
From: _____
Company: _____

CodeCheck Version: _____ Abraxas Part Number: _____

Operating System: _____ Platform: _____

C or C++ compiler: _____

Your phone number: _____

Your fax number: _____

Your Inter-Net address : _____

Please describe your problem. *If it is a syntax warning or fatal error, please try the suggestions found in Tech Note #6 on Troubleshooting before faxing in a Trouble Report Form!* It frequently helps to show us the relevant portion of a listing file, so that we can see the error message in its exact context. Make this listing file by running CodeCheck with the **–H**, **–M**, and **–D?** options. Do not use **–J**. The listing file created by CodeCheck will have the name check.lst.

Generally if you have a compile problem here is what will be asked.

1.) Never use the –R until you have verified that you can compile your source just like 'cc –c foo.c', e.g. "check foo.c'.

2.) If you have a problem with step #1 we ask that you send the generated check.lst [-L –M –H], and the dot-eye file. "cc –E foo.c > foo.c.i".

Never send proprietary code. Generally if there is a problem, its in the system headers, .e.g. "#include<iostream>,… If you cut & paste just the #include < system - include >'s from the top of your C/C++ source generally you can create a small c or c++ test case that doesn't include your proprietary code.

Most often problems are that #defines are missing, for instance MS-DEV C++ requires many explicit macros to emulate the /MT, /MD modes of operation. { .e.g. /MD requires –D_MD on the CodeCheck cmd-line. See MSDEV help-topic "/MD" for macro equivalent. }

All GCC compilers are different, gcc/g++ always requires CCP files for configuration. Years ago ALL system headers files on UNIX Clones were simply kept in /usr/include. This is the case no longer, today on any given GCC compiler installation dozen's of include search path's are required to emulate most g++ compilers. For G++ the simplest thing is to request our MKCCP tool that generates CCP file(s) automatically for gcc. This is advanced tool, your first CCP should always be written by us, so that you can have a working example as a basis. See the document Linux Standard Base with CodeCheck for a full description of the GCC processing issues.