CV-Suite:
The Plum Hall Validation Suite for C

Version 2025a August 2025

Your Feedback is Valued

Please feel free to contact me with any issues, errors, omissions, thoughts, … concerning the test cases and infrastructure in the Plum Hall test suites. The software is constantly updated with new test cases and infrastructure improvements. A new distribution is released in the month of August every year. Please contact me by email: dougteeple at plumhall2b.com.

New in cvs25a:

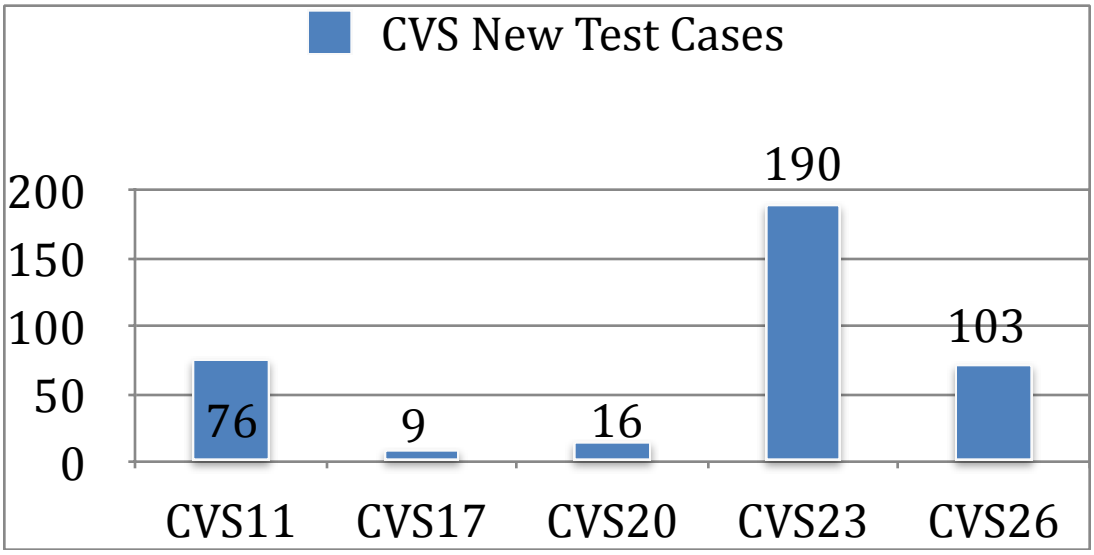
This release addresses many defect reports from customers in the 24a release and adds new tests for C26 proposed features as documented in publication n3301.pdf.

Version	ISO Document	__STDC_VERSION__	Comments
C11	ISO/IEC 9899:2011	201112L	
C17	ISO/IEC 9899:2018	201710L	AKA C18, C1X
C23	ISO/IEC 9899:2024	202311L	C2X
C26	ISO/IEC 9899:202y		C2Y

C Releases

New Test Cases

There are 103 new test cases documented in “coverage-c26.html”, in multiple directories. These new test cases predominantly pertain to the C26 proposed standards. The total number of test cases is now more than 4000. The new test case names are prefixed by c2y_. The test cases prefixed by c2x_ and c2y_ have a slightly different structure so that they can be run either as prior cases linked in to lang.c / lib.c or run standalone. This new feature (courtesy makefile magic) is useful during testing to run the tests individually through the runtest.sh script (e.g. runtest.sh c2y_5_2_1). The prior method of linking tests into lang.c and lib.c suffered from the fact that, if there was a failure in any test case, then lang/lib would fail to link, and no test results could be seen.



There are new test cases in a new category, which is by functionality:

c2y_23.c
c2y_26.c
c2y_MISRA_C23.c
c2y_ISO_26262.c
c2y_threads.c
c2y_lambda.c
c2y_embedded.c
c2y_freestanding.c
c2y_ub.c
c2y_annex_b.c
c2y_annex_f.o
c2y_annex_g.o
c2y_annex_h.o
c2y_annex_k.o
c2y_annex_d.o
c2y_annex_d.c
c2y_annex_e.c
c2y_annex_k.c
c2y_annex_j.c
c2y_annex_h.c
c2y_annex_g.c
c2y_annex_f.c
c2y_appendix_b.c

Each test case tests for specific functionality, alluded to by the test name.

Running the Test Suite

It is very important that you review envsuite(.bat), flags.h, compiler-flags.h and compiler-setup.bat to choose the correct settings for your compiler.

envsuite(.bat) is a script which is basically a large case statement. The cases are settings for different compilers. Common compilers are available in the script. If your compiler is not represented, you can use the existing implementations as a guide. envsuite is called in such a way that it instantiates environment variables used by the build script to run the test cases.

flags.h is a header file included by each test case. It defines flags which determine the standards year to test against and features that should be tested for the corresponding standards year.

compiler-flags.h is another header included by each test case. It defines specific flags for each compiler. If your compiler is not represented, add it using existing cases as a guide. The flags are very restricting as shipped. The reason is that is the only way for the tests actually to return any results, particularly for newer standards years, for which few, if any of the features tested actually compile. So, after an initial run to get basic results, you may see that many test cases are not as skipped. Over time, remove the restraint flags to test newer standards features.

compiler-setup.bat is a Windows-only script that should be run after envsuite.bat to set up specific compiler environment variables. Modify as required (but don’t forget to run this script after envsuite.bat).

Version 2024a August 2024

Your Feedback is Valued

Please feel free to contact me with any issues, errors, omissions, thoughts, … concerning the test cases and infrastructure in the Plum Hall test suites. The software is constantly updated with new test cases and infrastructure improvements. A new distribution is released in the month of August every year. Please contact me by email: dougteeple at plumhall2b.com.

New in cvs24a:

This release addresses many defect reports from customers in the 23a release and adds new tests for C26 proposed features.

Version	ISO Document	__STDC_VERSION__	Comments
C11	ISO/IEC 9899:2011	201112L	
C17	ISO/IEC 9899:2018	201710L	AKA C18, C1X
C23	ISO/IEC 9899:2024	202311L	C2X
C26			C2Y

C Releases

New Test Cases

There are 30 new test cases documented in “newcases-cvs23a-cvs24a.txt”, in multiple directories. These new test cases predominantly pertain to the C26 proposed standards. The new test case names are prefixed by c2x_ and c2y_.

The 24a release represents 3+ years of test case bug fixing, infrastructure improvements, and new test cases for C17, C20, C23, C26, language and library enhancements. There are many improvements in enhancing the test cases themselves, scripting and enhancing the reporting of the results, through the new html interfaces for reporting coverage, commentary on the intent of the test cases and improved standards conformance reporting.

The file compiler-flags.h contains defines for common compilers. Modify these settings if your compiler is implemented in the list, our add custom settings for your compiler. Also review envsuite in detail. Some suggested settings are included for common compilers. Use these settings to create the appropriate build environment for the version of the compiler that you wish to test.

Version 2023a August 2023

New in cvs23a:

This release addresses many defect reports from customers in the 22a release and adds new tests for C20 and C23 features. As of this release support for older C versions prior to C11 is dropped.

Version	ISO Document	__STDC_VERSION__	Comments
C11	ISO/IEC 9899:2011	201112L	
C17	ISO/IEC 9899:2018	201710L	AKA C18, C1X
C20			AKA C2X
C23			Proposed

C Releases

New Test Cases

There are 30 new test case files, documented in “newcases-cvs23a-cvs24a.txt”, in multiple directories. These new test cases predominantly pertain to the C23 and C26 standards.

Bug Fixes

The 23a update release represents 3+ years of test case bug fixing, infrastructure improvements, and new test cases for C17, C20, C23, C26 (proposed) language and library enhancements. There are many improvements in enhancing the test cases themselves and also enhancing the reporting of the results, through the new html interfaces for reporting coverage, commentary on the intent of the test cases and improved standards conformance reporting.

Infrastructure

Installers are available on the PlumHall server for Linux (installPH.sh) and Windows (installPH.bat). These installers greatly simplify installing the PlumHall distributions in a standard layout as described below. Download the installers from the plumhall2b.com server and create the default installations. The installers are customized for each customer:

```
ftp plumhall2b.com
Connected to plumhall2b.com.
220---------- Welcome to Pure-FTPd [privsep] [TLS] ----------
Name (plumhall2b.com:doug): OscarWilde1854
331 User OscarWilde1854 OK. Password required
Password: ************
passive
get installPH.sh
get installPH.bat
quit

~/installPH.sh --help
Download, check the MD5 hash and install the PlumHall test suites in $HOME/PlumHall/
Options:
  --cvs=<version>       : install CVS version e.g. --cvs=CVS002
  --xvs=<version>       : install XVS version e.g. --xvs=XVS002
  --lvs=<version>       : install CVS version e.g. --lvs=LVS002
  --compiler=<name>     : brief compiler name used to create directory
                          structure, e.g. gcc, edg, clang, etc
  --PW=<zip password>   : zip password for distribution
  --login=<login name>  : login name given in download instructions, e.g. techcontactname
  --username=<username> : suffix to user name given in download instructions,
                          e.g. techcontactname8345
  --keep                : do not delete existing directories before unpacking the distributions.
  --verbose             : chatty
  --help                : help me if you can…

Note: uses scp to securely copy the distributions from the plumhall2b.com server, zip to unpack the
distribution and md5sum to calculate the MD5 hash.
```

Executing the scripts will download your distributions and check the MD5 sums. If the sums do not match the scripts will exit with an error, please contact PlumHall should this occur. If the MD5 sums are correct then compressed files will be expanded and installed in the standard directory structure.

If you have trouble with the install scripts, you may enter the commands:

```
ftp plumhall2b.com
login: OscarWilde1854@plumhall2b.com
passwd: *********
passive
get installPH.sh or get installPH.bat
get cvs23a-CVS000.tar.gz
get cvs23a-CVS000.tar.gz.md5
get cvs23a-CVS000.zip
get cvs23a-CVS000.zip.md5
…
bye
```

If you did a manual download you may then run the installer script with the option *--nodownload* to unpack, check the MD5 signatures and create and populate the standard directory structures. The installer extracts into a directory named ~/PlumHall/ by default. Please ensure that the md5sum utility is available
and verify that the MD5 sums compare.

The script will ask for your plumhall2b ftp password as part of the installation process.

The default folder naming convention is:

```
<Test Suite><PlumHall Release Year>-<Compiler Mnemonic>-c<Standards Year>/

e.g. cvs23a-gcc-c20/
```

For example, to create directories for each of the standards years C17 and C20, for compilers gcc and clang:

```
installPH.sh --stdyear=17 --stdyear=20 --compiler=gcc --compiler=clang
```

There are three main points of customization:

- flags.h for C/C++ version options,
- compiler-flags.h for compiler-specfic options and
- envsuite.sh (envsuite.bat) to customize the execution environment.

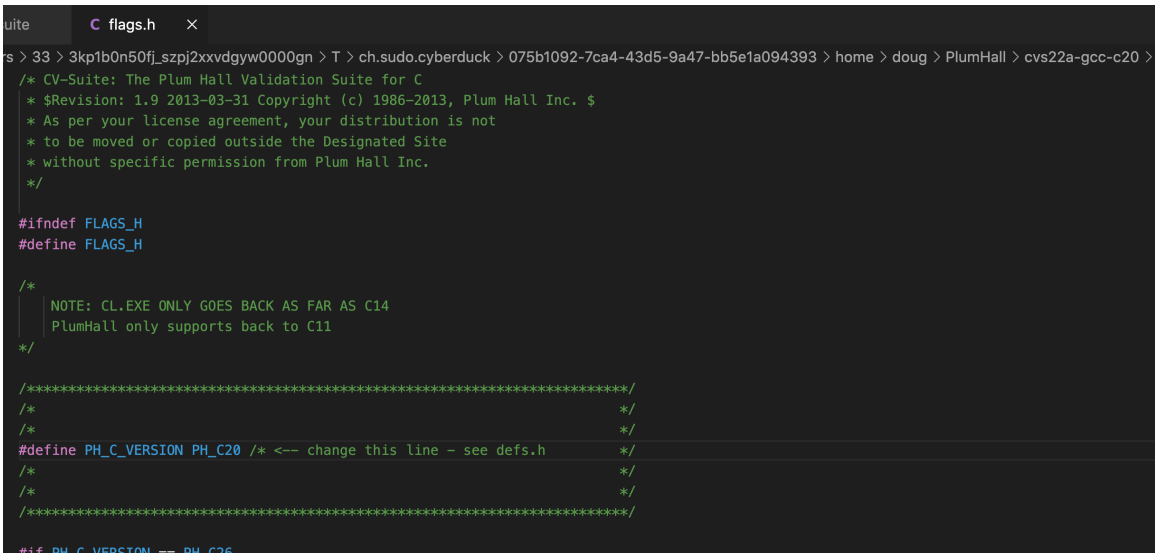
Some customization is possible by using envsuite command line options. For example: `envsuite.sh cc=g++-latest` sets the latest version of gcc to use. Type `envsuite.sh -h` for current arguments. Further customization requires editing envsuite.sh(.bat)

The `envsuite` script has been modified to more easily support a standard PlumHall directory structure and multiple compilers on the command line. The standard directory structure is:

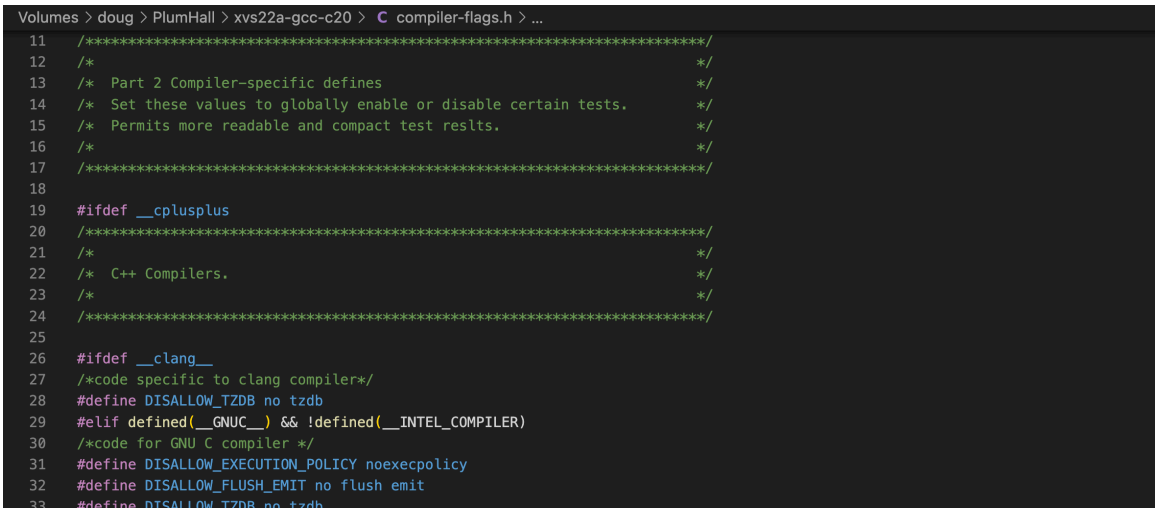
```
~/PlumHall/xvs23a-<cc>-c20/          build directory
~/PlumHall/xvs23a                    source directory
~/PlumHall/xvs23a-<cc>-c20-setup/    setup directory updated by script save-setup

~/PlumHall/lvs23a-<cc>-c20/
~/PlumHall/lvs23a/
~/PlumHall/lvs23a-<cc>-c20-setup/

~/PlumHall/cvs23a-<cc>-c20/
~/PlumHall/cvs23a/
~/PlumHall/cvs23a-<cc>-c20-setup/
```

where <*cc*> is gcc or clang, or cl on Windows. The script `createDestination.sh` is available to create and populates these default directories, though the `installPH` scripts do this by default. It takes a command line argument cc=<gcc | clang | cl> to create different build directories for multiple compiler testing. The scripts take arguments cc=gcc or cc=gcc-10 to cc=clang-12 as examples. **`envsuite` will need editing for your particular environment.** PH_C26 is set as the default release in flags.h and envsuite.

The file flags.h customizes for C standards release version:



Customization of flags.h

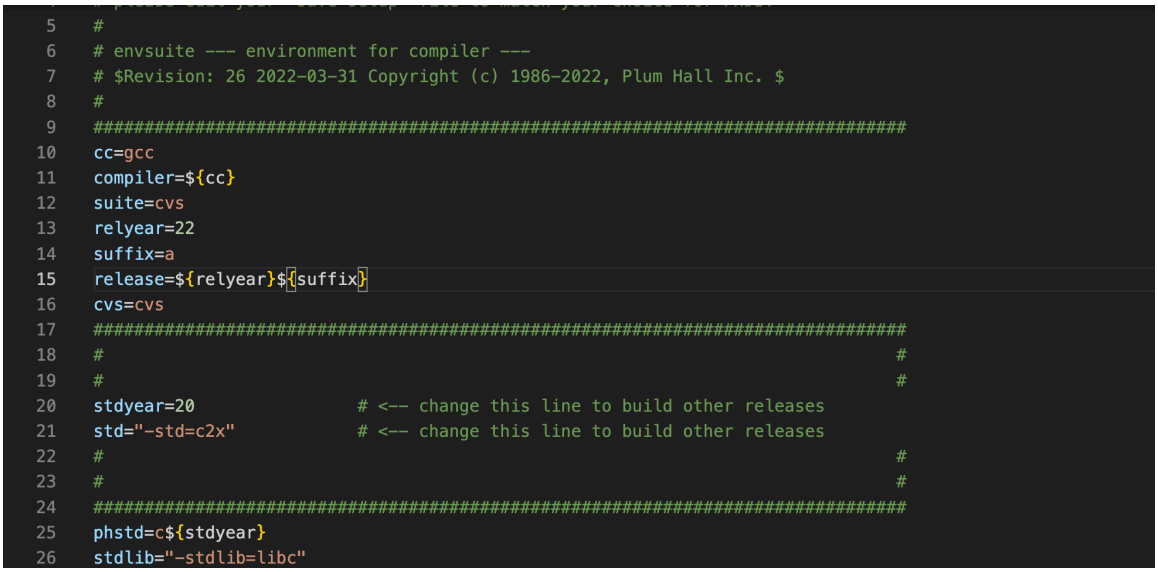


Customization of compiler-flags.h

The file compiler-flags.h allows for setting flags specific to a particular compiler. These flags are often set to get around compile errors which prevent viewing overall results. For example lang.c and lib.c link in relevant test case object files. If a compile of a particular test fails, none of the results of the other tests can be seen.

The file compiler-flags.h contains defines for common compilers. Modify these settings if your compiler is implemented in the list, our add custom settings for your compiler. Also review envsuite in detail. Some suggested settings are included for common compilers. Use these settings to create the appropriate build environment for the version of the compiler which you wish to test.

It is very important that you review envsuite(.bat), flags.h and compiler-flags.h to choose the correct settings for your compiler.



```
#
# envsuite --- environment for compiler ---
# $Revision: 26 2022-03-31 Copyright (c) 1986-2022, Plum Hall Inc. $
#
#############################################################################
cc=gcc
compiler=${cc}
suite=cvs
relyear=22
suffix=a
release=${relyear}${suffix}
cvs=cvs
#############################################################################
#                                                                           #
#                                                                           #
stdyear=20              # <-- change this line to build other releases
std="-std=c2x"          # <-- change this line to build other releases
#                                                                           #
#                                                                           #
#############################################################################
phstd=c${stdyear}
stdlib="-stdlib=libc"
```

Customization of envsuite(.bat)

The release numbers in flags.h and envsuite MUST be kept in sync.

The build system itself has been enhanced. In prior releases adding a test case required hand editing multiple different makefiles and scripts. In this release this is no longer required, the makefiles and script automatically adjust to addition/deletion of test cases.

For example on Linux:

```
. ./envsuite cc=gcc-latest
```

At the end of each buildmax build the following html files are created:

```
coverage-c20.html
commentary-c20.html
ctests-c20.html
conform-c20.html
conform-ctests-c20.html
report-c20.html
```

The file conform-ctests-cxx shows a summary of successful tests and those with issues:
The links to the source file, the output log, and the error log are all active and viewed as html.

The value in the **Expected** column is the number of test cases, where Expected = Actual + Errors + Faults + Aborts. The **Actual** column is the sum of the number of test results that matched expected values/behavior plus the number of skipped test cases. The value in the **Skipped** column is the number of skipped test cases. The value in the **Errors** column is the sum of the number of test cases that meet one of the following conditions:

- One or more unexpected values are returned in the test items.
- A compile error occurred, when the test file was compiled.
- An execution error occurred, when the test was executed.

CSuite Conformance c20 cvs22a - gcc 11.0.1 20210 - Linux - Sat Jan 28 14:56:59 2023

conform | C Conformance Tests.

ERROR in ch65c.c at line 364: (1) != (4) ch65c.c				Structures ..... all sorts of tests of structures and unions, DR452 Effective Type in Loop Invariant - C17, DR499 Anonymous structure in union C17, DR413 struct initialization			
ERROR in ch65c.c at line 365: (67305985) != (4) ch65c.c				Structures ..... all sorts of tests of structures and unions, DR452 Effective Type in Loop Invariant - C17, DR499 Anonymous structure in union C17, DR413 struct initialization			
166	164	98%	1	2	0	0	See results in the output log.
Segfault or fatal compile error ch72.c				7.2 - Diagnostics C99/C11 N2829: Make assert() macro user friendly for C and C++			
ERROR in ch7_12.c at line 2304 ch7_12.c				C90 7.5 C99/C11 7.12 - Mathematics, DR410 ilogb inconsistent with lrint, lround - C17, DR415 Missing divide by zero entry in Annex J - C17, D mtx_trylock should be allowed to fail spuriously, DR409 "A range error occurs if x is too large." is misleading - C20, DR473 "A range error oc too large." is misleading - C20, DR500 Ambiguous specification for FLT_EVAL_METHOD, FPDR21 printf of one-digit character string, N2283 p one-digit character string, FPDR24 remainder NaN case, N2186 Alternative to N2166, N2271: CR for pow divide-by-zero case, N2359: part 2 (r WANT macros from numbered clauses) and part 3 (version macros for changed library clauses), N2751: signbit cleanup with typo fix - C2x N Normal and subnormal classification N2797: *_HAS_SUBNORM==0 implies what? - C23 N2993: Make *_HAS_SUBNORM be obsolescent - C2			
ERROR in ch7_12.c at line 2308 ch7_12.c				C90 7.5 C99/C11 7.12 - Mathematics, DR410 ilogb inconsistent with lrint, lround - C17, DR415 Missing divide by zero entry in Annex J - C17, D mtx_trylock should be allowed to fail spuriously, DR409 "A range error occurs if x is too large." is misleading - C20, DR473 "A range error oc too large." is misleading - C20, DR500 Ambiguous specification for FLT_EVAL_METHOD, FPDR21 printf of one-digit character string, N2283 p one-digit character string, FPDR24 remainder NaN case, N2186 Alternative to N2166, N2271: CR for pow divide-by-zero case, N2359: part 2 (r WANT macros from numbered clauses) and part 3 (version macros for changed library clauses), N2751: signbit cleanup with typo fix - C2x N Normal and subnormal classification N2797: *_HAS_SUBNORM==0 implies what? - C23 N2993: Make *_HAS_SUBNORM be obsolescent - C2			
171	162	94%	6	8	1	0	See results in the output log.
1642	1642	100%	0	0	0	0	See results in the output log.
2413	2413	100%	0	0	0	0	See results in the output log.

negtests | Negative Tests - tests that should fail.

	Expected	Actual	Overall	Skipped	Errors	Faults	Aborts	Test Description
	4888	4855	99%	7	32	1	0	

conform-ctests-cxx.html

The value in the **Abort** column is the number of test cases that and abort occurred. The value in the Faults column is the number of tests that meet one of the following conditions:

- An uncaught exception occurred when the test was executed.
- An internal error occurred when the test file was compiled.
- Unknown or unreported test results.

The links to the .out log file and .cpp source file help to quickly find what the issue is and where.

The “t***.out” log filename in column 1 is a link to the actual output log of test result summaries for the entire test directory. The center column shows errors linked to the “<testfilename>.clg” compiler log file showing compile errors.

lang.out

Output Log

```
/*****************************************************************\
 ** Testing conformance of C Language Syntax and Semantics       **
 ** (C) 1986-2022 Plum Hall Inc                                  **
\** CONFORM Version 2022a  2022-11-24                            **/

This copy of the Plum Hall Validation Suite for C (tm) is cvs22a-CVS002
licensed solely for the use of employees of PlumHall
for use within a two-mile radius of Plum Hall 67-1185 Mamalahoa Hwy Unit D104, PMB 372 Kamuela HI 96743 USA
 or a single secure cloud site.
The Primary Contact person at this site is Thomas Plum


Technical support is provided by  Plum Hall Inc,
plum@plumhall.com.  The software is copyrighted
and proprietary to Plum Hall Inc, and by license is restricted to the
two-mile-radius site.  The reports it produces (such as this one) are
not for distribution outside your organization.  Consult your license
for any special provisions.   We appreciate your ideas and questions.

Version Dating: If this software is executed after the date below, it
is probably out-of-date. In that case, consult with your site Primary
Contact to obtain an up-to-date copy.

Expiration Date: 2021-12-31

===== ch62
***** Reached first test *****
#PASSED:  ch62a
#PASSED:  ch621
#PASSED:  ch622
#PASSED:  ch623
#PASSED:  ch624
```

Output Log File

The compiler error file shows any compiler error output:

ch66.c

Compile Error Report

```
1.    ch66.c:107:1  warning: variably modified ‘array_or_varlenarray’ at file scope
2.    ch66.c:210:15  warning: unsigned conversion from ‘int’ to ‘unsigned char:1’ changes value from ‘2’ to ‘0’ [-Woverflow]
3.
```

```
1.
2.  /* The Plum Hall Validation Suite for C
3.   * Unpublished copyright (c) 1986-2005, Plum Hall Inc.
4.   * DATE: 2005-03-31
5.   * As per your license agreement, your distribution is not
6.   * to be moved or copied outside the Designated Site
7.   * without specific permission from Plum Hall Inc.
8.   */
9.
10.  #include "defs.h"
11.
```

ch66.clg.html

The test source file:

ch72.c

```
1.
2.  /* The Plum Hall Validation Suite for C
3.   * Unpublished copyright (c) 1986-2013, Plum Hall Inc.
4.   * DATE: 2013-03-31
5.   * As per your license agreement, your distribution is not
6.   * to be moved or copied outside the Designated Site
7.   * without specific permission from Plum Hall Inc.
8.   */
9.
10.  #undef LIB_TEST
11.  #define LIB_TEST 1
12.
13.  #include "defs.h"
14.
15.  #ifndef SKIPch72
16.
17.  /* ch72 7.2 - Diagnostics <assert.h> C99/C11
18.         N2829: Make assert() macro user friendly for C and C++
19.  */
20.
21.  #include <assert.h>
22.  void ch72(void);
23.  static void ch721(void);
24.  void ch721b(void);
25.
26.  /* NDEBUG is not defined by <assert.h>. Force an error if it is */
27.  #if defined(NDEBUG)
28.         complain(__LINE__);
29.  #endif
30.
31.  void ch72()
```

Test Source File

The `make-commentary` script creates an html file (commentary-cxx-gcc.html for example) that shows a brief commentary of the purpose of each test case by folder name and test name:

CSuite c20 Commentary cvs22a - gcc 11.0.1 20210 - Linux - Sat Jan 28 14:56:59 2023

conform | C Conformance Tests.

ch62.c	C90 6.1.2.1 C99/C11 6.2.1 Scopes of Identifiers, Labels followed by declarations and }, DR445 Issues with alignm
ch62.c	C90 6.1.2.1 C99/C11 6.2.1 Scopes of Identifiers, Labels followed by declarations and }, DR445 Issues with alignm
ch63.c	C90 6.2 C99/C11 6.3 Conversions.
ch63.c	C90 6.2 C99/C11 6.3 Conversions.
ch64.c	C90 6.1 C99/C11 6.4 Lexical Elements.
ch64.c	C90 6.1 C99/C11 6.4 Lexical Elements.
ch65a.c	C90 6.3 C99/C11 6.5 Expressions.
ch65c.c	Structures ..... all sorts of tests of structures and unions,
ch65c.c	Structures ..... all sorts of tests of structures and unions,
ch65c.c	Structures ..... all sorts of tests of structures and unions,
ch65c.c	Structures ..... all sorts of tests of structures and unions,
ch65b.c	DR415 Missing divide by zero entry in Annex J.2 - C17,
ch65b.c	DR415 Missing divide by zero entry in Annex J.2 - C17,
ch65a.c	C90 6.3 C99/C11 6.5 Expressions.
ch67.c	C90 6.5 C99/C11 6.7 Declarations.
ch67b.c	BITFIELDS - make sure that all operators work with bitfields.
ch67b.c	BITFIELDS - make sure that all operators work with bitfields.
ch67.c	C90 6.5 C99/C11 6.7 Declarations.

commentary-ctests-cxx.html

The filenames are links which will open the files for viewing in the html browser.

The `make-coverage` script generates the html file coverage-cvs24a.html which shows for each C release, the Defect Report number, the directory test case file and a brief description of the Defect Report. This is useful to find which directories and test cases address a particular feature introduced by the Defect Report.

CSuite Coverage cvs22a - gcc 10.2.0 - Linux - Sat Jan 28 14:56:59 2023

CVS11

CVS17 Click to view CVS17.

conform	dr445	Issues with alignment in C11, part 2	ch62_main.c	ch62_main C90 6.1.2.1 C99/C11 6.2.1 Scopes of Identifiers, Labels followed by de C17
conform	dr406	Visible sequences of side effects are redundant	c11_7_17c.c	c11_7_17c C11 7.17.7 Atomics <stdatomic.h>, Operations on atomic types. effects 7_1776/23, effects of atomic_load() and atomic_load_explicit(, DR406: Visible seq atomic_compare_exchange: What does it mean to say two structs compare equal _Decimal64, and _Decimal128), DR453: Atomic flag type and operations C23
conform	dr431	atomic_compare_exchange: What does it mean to say two structs compare equal?	c11_7_17c.c	c11_7_17c C11 7.17.7 Atomics <stdatomic.h>, Operations on atomic types. effects 7_1776/23, effects of atomic_load() and atomic_load_explicit(, DR406: Visible seq atomic_compare_exchange: What does it mean to say two structs compare equal _Decimal64, and _Decimal128), DR453: Atomic flag type and operations C23
conform	dr453	Atomic flag type and operations	c11_7_17c.c	c11_7_17c C11 7.17.7 Atomics <stdatomic.h>, Operations on atomic types. effects 7_1776/23, effects of atomic_load() and atomic_load_explicit(, DR406: Visible seq atomic_compare_exchange: What does it mean to say two structs compare equal _Decimal64, and _Decimal128), DR453: Atomic flag type and operations C23
conform	dr491	Concern with Keywords that Match Reserved Identifiers	c11_7_26f.c	
conform	dr462	Clarifying objects accessed in signal handlers	ch7_14.c	ch7_14 C90 7.7 C99/C11 7.14 - Signal handling. Both signal() and raise() are teste
conform	dr428	runtime-constraint issue with sprintf family of routines in Annex K	ch7_24d.c	
conform	dr441	Floating-point issues in C11 from PDTS 18661-1 UK review, Issue	ch77.c	

coverage-cxx.html

The make-report script generates a table showing all files with the associated commentary:

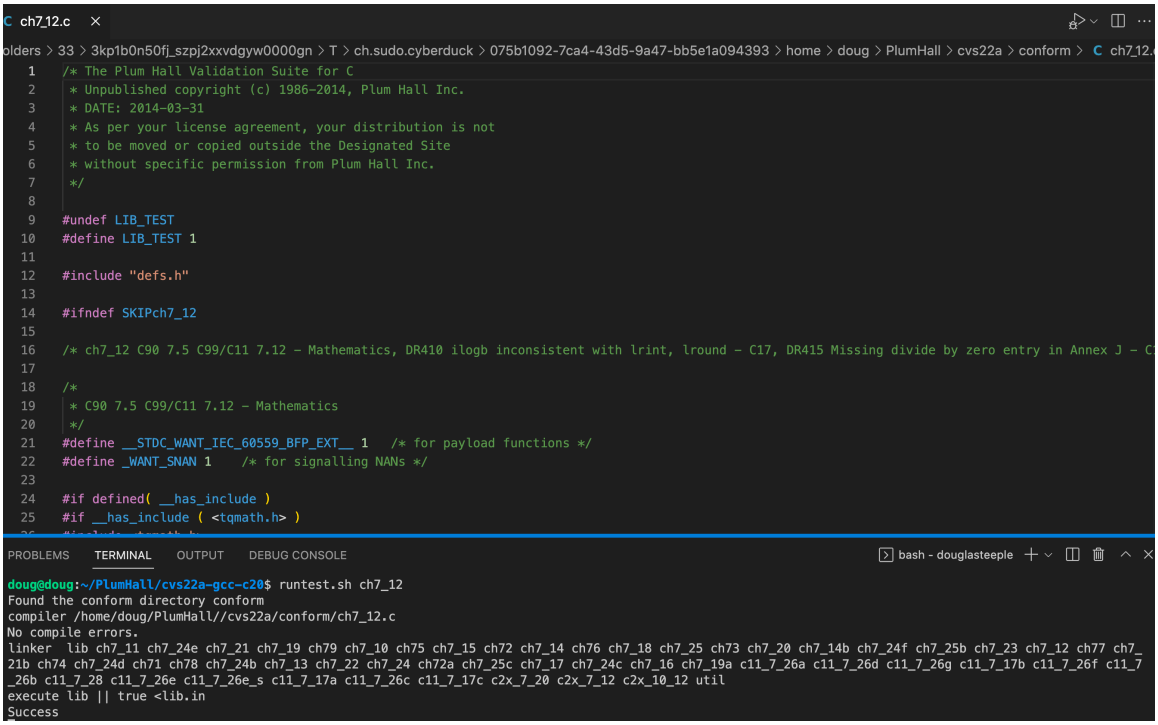
HTML Report of Commentary from All Sources - c20

cvs-22a	
bench	
bench.c	- driver for Plum Hall benchmarks
f3.c	- call the virtual functions (of unknown type)
conform	
ch65a.c	C90 6.3 C99/C11 6.5 Expressions. DR423 - underspecification for qualified rvalues, see also n05.in for negtests, DR444 Issues with alignment in C11, part 1 - C17, DR480 cnd_wait and cnd_timewait should allow spurious wake-ups, DR481 Controlling expression of _Generic primary expression, N2293 Alignment requirements for memory management functions, N2381 Unnamed parameters in function definitions, Identical cvr-qualifications for array types and their element types
ch62_main.c	C90 6.1.2.1 C99/C11 6.2.1 Scopes of Identifiers, Labels followed by declarations and }, DR445 Issues with alignment in C11, part 2 - C17
ch7_24d.c	7.24.4 General wide string utilities 4.6.3 C99 DR428 runtime-constraint issue with sprintf family of routines in Annex K - C17
ch66.c	Constant expressions C90 6.4 C99/C11 6.6 N2713: Integer Constant Expression
c2x_7_21.c	7.21 NN2571: snprintf et al - C2x, N2349: the memccpy function - C2x, N2353: the strdup and strndup functions - C23 N2826: Add annotations for unreachable control flow proposal for addition to C23 and TS 6010 7.21.1 The unreachable macro - C23
ch76.c	Floating point environment <fenv.h>, N2124 rounding direction macro FE_TONEARESTFROMZERO - C2x, N2319 rounding direction macro FE_TONEARESTFROMZERO - C2x
c11_7_17a.c	C11 7.17.1 Atomics <stdatomic.h>, DR453 Atomic flag type and operations, DR485 Problem with the specification of ATOMIC_VAR_INIT

report-cxx.html

Again the file names are links for convenient browsing of the test case suite. All of these html documents are produced dynamically from the source as the last steps in the `buildmax` script.

There is a new script `runtest.sh(.bat)` which given a test identifier will find that file in the source directory and execute just that test. It is useful for debugging test cases. Here is an example of usage:



Please let me know your thoughts and suggestions: doug@plumhall2b.com.

C20 Language/ Library Features - ISO/IEC 9899:202x

Feature	Paper	Addressed	Test Cases
volatile semantics for lvalues	DR 476		
c16rtomb() on wide characters encoded as multiple char16_t	DR 488		c11_7_28.c
Part 1: Alignment specifier expression evaluation	DR 494		ch65a.c
"white-space character" defined in two places	DR 497		ch7_252h.h
Anonymous structure in union behavior	DR 499		ch65c.c
Ambiguous specification for FLT_EVAL_METHOD	DR 500		ch7_12.c
make DECIMAL_DIG obsolescent	DR 501		ch77.c
changes for obsolescing DECIMAL_DIG	FPDR20		c2x_7_20.c
printf of one-digit character string	FPDR21, N2283		ch7_12.c
llquantexp invalid case	FPDR23		c2x_7_12.c
remainder NaN case	FPDR24		ch7_12.c
totalorder parameters	FPDR25		c2x_7_12.c
rounding direction macro FE_TONEARESTFROMZERO	N2124, N2319		ch76.c
Alternative to N2166	N2186		ch7_12.c
type generic **cbrt** (with editorial changes)	N2212		ch7_22h.h
Clarifying the **restrict** Keyword v2	N2260		ch67.c
Harmonizing _assert with C++	N2265		n07.in
nodiscard attribute	N2267		c2x_6_711a.c
maybe_unused attribute	N2270		c2x_6_711a.c
CR for pow divide-by-zero case	N2271		ch7_12.c
Alignment requirements for memory management functions	N2293		ch65a.c
preprocessor line numbers unspecified	N2322		ch6_10.c
DBL_NORM_MAX etc	N2325		ch77.c
floating-point zero and other normalization	N2326		ch77.c
deprecated attribute	N2334		c2x_6_711a.c
strftime, with 'b' and 'B' swapped	N2337		ch7_23.c
error indicator for encoding errors in fgetwc	N2338		ch7_24b.c
editors, resolve ambiguity of a semicolon	N2345		c11_7_26a.c
the memccpy function	N2349		ch7_21.c
defining new types in offsetof	N2350		n07.in
the strdup and strndup functions	N2353		ch7_21.c
update for payload functions	N2356		ch7_12.c
no internal state for mblen	N2358		
part 2 (remove WANT macros from numbered clauses) and part 3 (version macros for changed library clauses)	N2359		ch7_12.c
The fallthrough attribute	N2408		c2x_6_711a.c
Two's complement sign representation for C2x	N2412		ch7_10.c
Section 6: Add time conversion functions that are relatively thread-safe	N2417		c11_7_26d.c
Adding the u8 character prefix	N2418		ch64.c, ch67.c, negtests/n01.in
Remove support for function definitions with identifier lists	N2432		n07.in

C17 Language/Library Features ISO/IEC 9899:2018

Feature	Paper	Addressed	Test Cases
realloc with size zero problems	DR 400		flags.h, conform/ch7_20.c
"happens before" cannot be cyclic	DR 401		conform/c11_717b.c
memory model coherence is not aligned with C++11	DR 402		conform/ch_67.c, conform/c11_7_26b.c
malloc() and free() in the memory model	DR 403		conform/ch7_20.c
joke fragment remains in a footnote	DR 404		
mutex specification not aligned with C++11 on total order	DR 405		conform/c11_7_26e.c
Visible sequences of side effects are redundant	DR 406		conform/c11_7_17c.c
SC fences do not restrict modification order enough	DR 407		conform/c11_7_17b.c
ilogb inconsistent with lrint, lround	DR 410		conform/ch7_12.c
#elif	DR 412		conform/ch6_10.c
typos in 6.27 threads.h	DR 414		
Missing divide by zero entry in Annex J.2	DR 415		conform/ch65b.c
Proposed defect report regarding tss_t	DR 416		conform/c11_7_26g.c
Missing entries in Annex J	DR 417		
What the heck is a "generic function"?	DR 419		conform/c11_7_26b.c
underspecification for qualified rvalues	DR 423		
G.5.1: -yv and -x/v are ambiguous	DR 426		
runtime-constraint issue with sprintf family of routines in Annex K	DR 428		ch7_18.c, ch7_19.c, ch7_20.c, ch7_21.c, ch7_23.c, ch7_24c.c, ch7_24d.c, ch7_24f.c
Should gets_s discard next input line when (s == NULL) ?	DR 429		conform/ch7_19.c
getenv_s, maxsize should be allowed to be zero	DR 430		conform/ch7_20.c
atomic_compare_exchange: What does it mean to say two structs compare equal?	DR 431		conform/c11_7_17c.c
Issue with constraints for wide character function	DR 433		conform/ch7_24f.c, conform/ch7_20.c
Missing constraint w.r.t. Atomic	DR 434		conform/ch67.c, negtests/n07.in
Request for interpretation of C11 6.8.5#6	DR 436		
clock overflow	DR 437		conform/ch7_23.c
ungetc / ungetwc and file position after discarding push back	DR 438		conform/ch7_24b.c
Issues with the definition of “full expression”	DR 439		conform/ch67.c
Floating-point issues in C11 from PDTS 18661-1 UK review, Issue 2	DR 441		conform/ch77.c
Issues with alignment in C11, part 1	DR 444		negtests/n07.in, conform/ch65a.c
Issues with alignment in C11, part 2	DR 445		conform/ch62.c, conform/ch7_17.c
Boolean from complex	DR 447		conform/ch65b.c
What are the semantics of a # non-directive?	DR 448		conform/ch6_10.c
tmpnam_s clears s[0]	DR 450		conform/ch7_19.c
Effective Type in Loop Invariant	DR 452		conform/ch65c.c
Atomic flag type and operations	DR 453		conform/c11_7_17.c
The ctime_s function in Annex K defined incorrectly	DR 457		
ATOMIC_XXX_LOCK_FREE macros not constant expressions	DR 458		conform/c11_7_17b.c
atomic_load missing const qualifier	DR 459		
aligned_alloc underspecified	DR 460		conform/ch7_20.c
Clarifying objects accessed in signal handlers	DR 462		
Clarifying the Behavior of the #line Directive	DR 464		conform/ch6_10.c
Fixing an inconsistency in atomic_is_lock_free	DR 465		conform/c11_7_17b.c
strncpy_s clobbers buffer past null	DR 468		conform/ch7_21.c
mtx_trylock should be allowed to fail spuriously	DR 470		
Complex math functions cacosh and ctanh	DR 471		conform/ch73.c, conform/ch7_22j.h,
Introduction to complex arithmetic in 7.3.1p3 wrong due to CMPLX	DR 472		
"A range error occurs if x is too large." is misleading	DR 473		conform/ch7_12.c
Misleading Atomic library references to atomic types	DR 475		
nan should take a string argument	DR 477		conform/negtests/n01.in #041
cnd_wait and cnd_timewait should allow spurious wake-ups	DR 480		
Controlling expression of _Generic primary expression	DR 481		conform/ch65a.c
Problem with the specification of ATOMIC_VAR_INIT	DR 485		conform/c11_7_17a.c
timespec vs. tm	DR 487		
Concern with Keywords that Match Reserved Identifiers	DR 491		conform/negtests/n01.in #42

C11 Language Features - ISO/IEC 9899:2011

Feature	Addressed	Test Cases
gets()		
Atomic objects (_Atomic)		c11_7_17a.c, c11_7_17b.c, c11_7_17c.c, c11_7_17c.c, c11_7_26b.c, ch62.c, ch67.c, ch7_14b., ch7_14.c, negtests/n07.in
Thread local storage (_Thread_local)		c11_7_17a.c, c11_7_26a.c, negtests/n07.in
Alignment query (_Alignof)		ch62.c, ch65a.c, ch66.c, ch7_14.c, negtests/n05.in
Alignment strengthening (_Alignas)		ch67.c, ch7_14.c, negtests/n07.in, negtests/n07.in
Over-aligned types		ch67.c
u/U character constants		ch64.c, ch67.c
u8/u/U string literals		ch64.c, ch67.c
Generic selection expressions (_Generic)		ch63.c, ch63.c, ch65a.c, ch65b.c, negtests/n05.in
Non-returning functions (_Noreturn)		c11_7_26f, ch67.c, ch7_20.c, negtests/n01.in
Anonymous struct and union members		ch7.c
Fine-grained evaluation order		
Extending the lifetime of temporary objects		ch62.c
__assert		negtests/n07.in
__STDC_ANALYZABLE__		ch6_10.c
__STDC_LIB_EXT1__		ch6_10.c
__STDC_NO_ATOMICS__		ch6_10.c
__STDC_NO_COMPLEX__		ch6_10.c
__STDC_NO_THREADS__		ch6_10.c
__STDC_NO_VLA__		ch6_10.c

C11 Library Features - ISO/IEC 9899:2011

Feature	Addressed	Test Cases
<stdalign.h>		ch7_14.c
<stdatomic.h>		c11_7_17a.c, c11_7_17a.c, c11_7_17b.c, c11_7_17b.c, c11_7_17c.c, c11_7_17c.c, c11_7_26a.c, c11_7_26b.c, c11_7_26c.c, c11_7_26d.c, c11_7_26e.c, c11_7_26f.c, c11_7_26g.c
<stdnoreturn.h>		ch7_20.c
<threads.h>		c11_7_17b.c, c11_7_26a.c, c11_7_26a.c, c11_7_26b.c, c11_7_26b.c, c11_7_26c.c, c11_7_26c.c, c11_7_26d.c, c11_7_26d.c, c11_7_26e.c, c11_7_26e_s.c, c11_7_26f.c, c11_7_26g.c, ch7_23.c, ch7_23.c
<uchar.h>		c11_7_17a.c, c11_7_17c.c, c11_7_28.c, c11_7_28.c, ch7_24.c, ch7_24f.c
Atomic operation library		c11_7_17a.c, c11_7_17a.c, c11_7_17b.c, c11_7_17b.c, c11_7_17c.c, c11_7_17c.c, c11_7_26a.c, c11_7_26b.c, c11_7_26c.c, c11_7_26d.c, c11_7_26e.c, c11_7_26f.c, c11_7_26g.c
Thread support library		c11_7_17b.c, c11_7_26a.c, c11_7_26a.c, c11_7_26b.c, c11_7_26b.c, c11_7_26c.c, c11_7_26c.c, c11_7_26d.c, c11_7_26d.c, c11_7_26e.c, c11_7_26e_s.c, c11_7_26f.c, c11_7_26g.c, ch7_23.c, ch7_23.c
aligned_alloc()		ch7_20.c
char16_t		c11_7_17a.c, c11_7_17a.c, c11_7_17a.cs, c11_7_17b.c, c11_7_17c.c, c11_7_28.c, ch64.c, ch64.c, ch67.c, ch7_24f.c
char32_t		c11_7_17a.c, c11_7_17a.c, c11_7_17a.cs, c11_7_17b.c, c11_7_17c.c, c11_7_28.c, ch64.c, ch64.c, ch67.c, ch7_24f.c
mbrtoc16()		c11_7_28.c, ch7_24f.c
mbrtoc32()		c11_7_28.c, ch7_24f.c
c16rtomb()		c11_7_28.c, ch7_24f.c
c32rtomb()		c11_7_28.c, ch7_24f.c
quick_exit		ch7_20.c
at_quick_exit		ch7_20.c
Exclusive modes of fopen() and freopen() ("x")		ch7_19.c
Bounds checking functions		
gets_s		ch7_19.c
fopen_s		ch7_19.c
printf_s		ch7_19.c
strcpy_s		ch_21.c
wcscpy_s		ch7_24d.c
mbstowcs_s		ch7_20.c
qsort_s		ch7_20.c
get_env_s		ch7_20.c
timespec		c11_7_26c.c, c11_7_26d.c, c11_7_26d.c, c11_7_26e.c, c11_7_26e.c, c11_7_26f.c
timespec_get()		c11_7_26d.c
CMPLX(FIL)		ch65b.c
(FLTIDBLILDBL)_DECIMAL_DIG		ch77.c
(FLTIDBLILDBL)_TRUE_MIN		ch77.c
(FLTIDBLILDBL)_HAS_SUBNORM		ch7_12.c
Thread local errno		c11_7_26b.c

Historical Versions

NOTE: SOME OF THE INFORMATION BELOW IS RETAINED AS A HISTORICAL REFERENCE.

For example, while C99 may be referenced, it is no longer supported, the suites do not support any version prior to C11.

Version 2020b August 2021

New in cvs20b:

Changes have been made to accommodate the C20 and CXX20 flags:

- ch65a.c add C20 to #if
- defs.h, add CXX20 to #if's
- c99.h, line 120, add C20 to #if
- flags.h, line 465, added C20 so as to include <stdbool.h>
- ctflags.h, added #If CXX20 #include <stdatomic.h>

New in cvs18a:

If you SKIP various testcases (see below), your numbers may be less than you might expect; SKIP'd testcases may cause other testcases to be SKIP'd as well.

New in cvs15a:
We have added a new `make-summary` command, which will produce the appropriate `.sum` file. Also, we added `buildmax`, to build everything (incorporating which-standard and whether-freestanding).

If there are no "dots" in the filename, the `txtchk` command will expect to find its checksums in a "`.txtchk`" file, so it can now be invoked as simply

```
txtchk  -f  cvs15a
```

New in cvs14a:
We now have 3 standards for C: the original "C90" (for which Plum Hall has routinely included the "widechar and digraphs" amendment), then "C99", and now "C11". We have accommodated this multiplicity by providing different expected-results files (the "fs" expected-results are for "freestanding", with minimal library):

conform-c90.exp	conform-c90-fs.exp
conform-c99.exp	conform-c99-fs.exp
conform-c11.exp	conform-c11-fs.exp

The C11 standard added some new and interesting requirements upon test suites, namely the provision of 8 optional features. Each of these features can be absent, without any impact on conformance; but if the feature is present, then all requirements must be met.

The ideal that we have been working toward is that, having selected the appropriate standard, and determined whether "freestanding" tests are to be used, then a 100% score will indicate full conformance, and anything less indicates a conformance problem.

One method we used to get somewhat closer to this ideal is that cvs14a has tried to package each of the tests for optional C11 features into functions which test for one or more required C11 features.

If you're testing against C11, and you do say that you have the (by-now optional) C99 <complex.h> feature, you indeed will get 7743 successful test cases in lib.out. But if DISALLOW_C99_COMPLEX is defined, you would have seen several hundred SKIPPED messages, which would spoil the ideal 100% score.

A minor version of the same problem goes back several years; there are 4 tests that showed up as SKIP's when testing against the C90 standard: ch78 (for <inttypes.h>), ch7_16 (for <stdbool.h>), ch7_18 (for <stdint.h>), and ch7_22 (for <tgmath.h>).

Our solution to this puzzle involves printing a message "omitting feature-x" when we otherwise would have printed a SKIP message. The conformance-test scoring program (summary) will not notice the "omitting" messages, so they will not be scored as a SKIP. Please let us know how this new method works for your projects.

We have had to simplify some of the harnessing; the combinatorics are threatening to overwhelm everyone. We have made the simplifying assumption that if C90 is the target, then EGEN64 and PH_INT64 will not be defined. In other words, if you are testing to the C90 standard, then EGEN will be built in the 32-bit version. If this isn't what you want, you will have to edit "flags.h" by hand, to specify EGEN64. Furthermore, the scripts make-c90, make-c99, and make-c11 will automatically build all the TESTING targets iff you are not testing "freestanding" (the fs option). You can edit the make-c?? scripts if you want other behaviors.

Also see "**Testing the C features of a C++ Compiler, in Suite++ CONFORM/CTESTS"** below.

New in cvs13a:

The 2011 revision of the C standard (ISO/IEC 9899:2011) is colloquially known as C11. In C11, there are eight options: (1) Threads, (2) Atomics, (3) Variable Length Arrays (VLAs), (4) Complex numbers (as in C99), (5) IEC 559 floating-point semantics (Annex F – not yet tested), (6) IEC 559 complex (Annex G – not yet tested), (7) Bounds-checking library functions (Annex K), and (8) Analyzable semantics (Annex L – not yet tested). An implementation can conform to C11 without providing any of these optional features (whereas the C99 standard required VLAs and complex numbers). To reflect these options, we've added suffixes on some of the case numbers: atomics (AT), bounds-checking (BC), complex (CX), VLA(VLA). We've also re-structured the scoring, so that C11 optional tests will be executed and verified for all options selected (thus producing ERROR messages for any failing cases), but the total number of expected successes is indifferent to the selecting or un-selecting of options.

We've attempted to place anything user-configurable into flags.h or sdutil.h, and to make defs.h invariant across all environments; please let us know if we overlooked anything. In particular, flags.h will be the place to select whether you are testing against C90 (which actually includes Amendment 1 from 1995), or C99, or C11 (and your selection of the C11 options).

We have augmented some of our tools to better accommodate the diverse ways our clients use CV-Suite in their overall testing program. The summary tool has, for several years now, printed "**" next to a file-name if there are any "unexpected" testcase FAILs or SKIPs in that output file. This year we have added another feature to summary: if in an output file (for example, abc.out) there are no lines that match the expected "***** N successful" or "***** N error" format (e.g., output was interrupted by a core dump or segfault), then "!!" is printed next to the file-name. Furthermore, if there is a file in the current directory named expected-fails, then summary will now read that file, save each line, and if abc.out appears in the expected-fails file, summary will just score all its testcases as "skipped and expected", without printing any marks next to the file-name abc.out. Furthermore, individual negtests (for example, n05002) can be put into the expected-fails file, and when the file of negtests output is being read (n05.out, in this example), any failure on testcase n05002 will be scored as "failed and expected".

Some of you create a new PHDST folder for each combination of compiler, option-flags, machine target, optimization level, etc. Others use the same PHDST folder and run make clobber between test runs. If the summary tool finds a file in the current directory named option-flags, (for example, containing the string fpsimulation-O3), then every time summary searches expected-fails for a testcase name (such as n05002), it will search for "fpsimulation-O3:n05002".

We've decided recently that it's better to place setup information into a folder that's not underneath the PHDST folder; that way, to completely clean the PHDST folder we can just execute `rm -rf *` in that folder (or the equivalent in a graphical display like Windows Explorer). After some experimentation, we recommend using `$PHDST-setup` (or `%PHDST%-setup`) as a folder name for the place to store setup information between test runs. This is what is now provided in the `envsuite` (or `envsuite.bat`) startup script. We also provide a script named `save-setup` (or `save-setup.bat`). So, to get started, edit three files: `flags.h`, `envsuite`, and `save-setup`.

New in cvs12a: see "EXP Files (Expected Results)" re **conform-c99.exp** (etc.);change **C9X** to **C99** everywhere; **fulltest** is now **make-c99** (etc.); **dst.2** is **dst-win**; **dst.3** is **dst-ix**; also see "Running the Suite"
New in cvs11a: see "Floating-point Comparisons" below
New in cvs10a: various bug-fixes.
New in cvs09a: see "Bounds-Checking TR" below
New in cvs08a: see "Bounds-Checking TR", "Unicode Strings TR", and "Expected NEGTESTS", below.

Copyright Notice

Trademarks

License

Please take the time to read the license that your organization has signed. It is a legal document, and the restrictions apply to any persons using the product. Here is a brief summary:

- You may use the CV-Suite on any machine within a 2-mile radius of your Designated Site.
- Your Management Contact, or anyone designated by the Management Contact, may call Plum Hall for consultation and advice.
- You need to notify us if you designate a new Management Contact, or plan to change your Designated Site, or plan to change your company's name.
- The CV-Suite is proprietary, confidential, copyrighted software. You must protect its confidentiality with the same procedures you use to protect your own company's confidential information.
- You may not disclose the detailed results of running the Suite, except as permitted in the License.

You may not take any form of copies of the Suite away from the Designated Site.

Thomas Plum authored a series of articles in Dr. Dobb's about the C11 standard; see
http://www.drdobbs.com/cpp/232800444 (C11 overview, concurrency, etc.)
http://www.drdobbs.com/cpp/232901670 (C11 security, Annex K, Annex L)
http://www.drdobbs.com/cpp/240001401 (Alignment, Unicode, ease-of-use features, C++ compatibility)

or, slightly re-formatted, in
open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3631.pdf

OVERVIEW

The Plum Hall Validation Suite is a set of C programs for testing and evaluating C language compilers. Each section of the Suite within ***conform*** (standard conformance testing) builds on the correctness established by the previous section.

This manual will explain how each section of the Suite works, how to configure the tests for your system, and what assumptions are made about previous sections. The examples will illustrate the use of the Suite, and also demonstrate how some of the sections work.

This manual is configured for Release 2016a (March 2016). Any up-to-the-minute changes or corrections (if any) are to be found on the passworded Plum Hall support web site; refer to your support email from Plum Hall.

If you have never used the Plum Hall Validation Suite for C you need to read this manual. This is a large, extremely configurable suite of test programs, it can provide you with a very powerful testing environment, but it usually takes several hours to set up the first time.

If you get stuck, or have problems, don't hesitate to call Plum Hall for technical support, we want you to succeed with this project.

The Test Suite

The Plum Hall Validation Suite consists of the following sections:

CONFORM

This section tests basic conformance to the C Standard. by configuring the `defs.h` file appropriately. A compiler can be compared with older levels of C or tested for conformance to the ANSI/ISO Standard for C. The C functionality of a C++ compiler maybe tested by modifying the `flags.h` file in the destination directory.

OPTIONAL

This section tests for diagnosis of the *undefined behaviors* described in the C Standard. These diagnostics are not mandated by the Standard but their detection by a compiler is an indication of *Quality of Implementation*

BENCH

Provides a small CPU-time benchmark

TESTING

The remaining sections are all found in one sub directory named *testing*

EXIN

The EXecutive INterpreter is a script language processor. When it is built and passes its own test set, the script processing is used as a basic tool in subsequent sections of the Suite.

COVER

This section uses EXIN scripts to generate self checking C programs that test coverage of all permutations of operators and data types. This section produces approximately 300 MB of generated C source.

LIMITS

More EXIN scripts that are used to determine the size of certain compile-time limits (e.g., significant length of identifiers or how deeply include files may be nested).

EGEN

The Expression GENerator is a test program, written in C, which generates self-checking expressions of arbitrary complexity. It is the tool used by the STRESS section.

STRESS

Since it is impossible to test all possible legal expressions, a sampling approach is taken. Under the control of EXIN scripts, EGEN is used to generate complex self-checking expressions. These can be completely random, under the control of a basic expression template, or driven from an EGEN script file.

TOOLS

Tools for use in different *destination* directories are provided in the directory trees named dst-win and dst-ix. Each of these contains a sub-tree that matches the structure of the source directories in CONFORM and TESTING.

What You Need to Know and Do

In order to install and run CV-Suite, there are several things you need to know, and several things you need to be able to do. If you don't have this knowledge yourself, then you need to locate someone who knows these things and is able to provide you with the information.

- You need to know how to use a text editor on each system you will be using.
- You need to know the basics of how to write and execute "script"' (or "batch"') files on each system.
- You need to know how much free disk space is available on each system. Thirty megabytes (30 MB) is often enough. (Much more may be required for the complete installation under DOS or Windows on large-sector drives.) If you don't have much more, refer to the "Installing this Release" section later in this chapter; if you have less, refer to the "Resources" section later in this chapter for details.
- You need to know some C programming, to customize certain files and to understand the general meaning of the compiler diagnostics that may be produced by some of the nastier test cases.
- You need to know which compiler you are supposed to test, and what commands, arguments, environment settings, etc., are needed in order to invoke the compiler you're testing. (The compiler you're testing is called the target compiler.) You may also need to use a different compiler to compile the tool programs themselves. This is known as the host compiler, and it may have its own commands, arguments, environment settings, etc.
- Similarly, you need to know how to invoke the target linker and the host linker, to link the object-files produced by the compilers.
- Once the target compiler and target linker have produced an executable program to be tested, you need to know how to execute that executable program. On some systems this is almost trivial; on others it involves downloading from one machine to another, capturing output, networking the output back to the host machine, etc.

Running the Suite

There are many different modes in which you can use the Plum Hall Suites:

- Script or batch command files for the compiler linker, etc., or line-by-line individual commands.
- Using your own harness, or using `make` (This keeps objects and executables around for faster re-compile).
- Host compiling (host and target compiler are the same) or cross compiling (host and target are different).
- UNIX platform, or Win32 platform, or some other platform.
- Hosted implementation (with library support), or freestanding implementation (e.g. embedded system, minimal library).

We have packaged the C Suite (and the C++ Suite) so that any set of these choices can be made.

The 2011 C standard incorporates new features and the accumulated interpretations (defect reports) of the ISO C 1999 standard, as well as ; use the **buildmax** script, which will compare results against the **conform-c11.exp** (or **conform-c11-fs.exp** for "freestanding") expected-results file. Be sure that the **C11** macro is defined in your flags.h header.

The 1999 C standard incorporates new features and the accumulated interpretations (defect reports) and amendments of the original ANSI C 1989 (i.e. ISO C 1990); again, use the **buildmax** script, which will compare results against the **conform-c99.exp exp** (or **conform-c99-fs.exp**) expected-results file. Be sure that the **C99** macro is defined in your flags.h header.

Some clients and agencies have used only the C90 (plus Amendment 1) requirements, as of our most recent information. That covers only the features of the original ANSI/ISO C, plus ten years' worth of corrections for those features; again, use the **buildmax** script, which will compare results against the **conform-c90.exp** (or **conform-c90-fs.exp**) expected-results file, define the **C90** macro in your flags.h, and un-define the **C99** macro. Further, you should consider each specific DISALLOW flag in flags.h, and determine whether it should be set for your specific

implementation. For example, if you are testing a C90 implementation that omits the 1995 Amendment 1 features (like digraphs), you need to be sure that the "C90" portion of your flags.h specifies

```
#define DISALLOW_DIGRAPHS 1
```

CONFORM

The CONFORM section of the Suite tests a compiler for conformance to the C Standard.

The CONFORM Tests

The CONFORM section consists of five C programs that test all of the required features of the language, preprocessor, and libraries as follows:

`ENVIRON`	tests Section 5 of the Standard.
`LANG`	tests the basic language and preprocessor. It is organized according to the section numbers of the ANSI standard document, but will only test features according to the selected language level.
`PREC1,PREC2`	test operator precedence. All C language operators are tested in all possible pairs to test that the precedence is handled correctly.
`LIB`	tests the C library. The organization follows the ANSI/ISO Standard

Each sub-section of the standard has a corresponding function in the LANG or LIB program. Each program uses utility routines for checking that two integers are equal (`iequals`), that two addresses match (`aequals`), etc. Errors are reported by writing a message of the form:

```
ERROR in c5.c line 234: (12) != (13)
```

Each program prints a summary in this form:

```
***** Reached first test *****
***** 999 successful tests in LANG *****
***** 2 errors found in LANG *****
***** 3 remarks found in LANG *****
***** 3 skipped sections in LANG *****
```

Skipped Sections

A "skipped" section results from compiling with a flag such as `SKIPch62` which causes all tests in `ch62.c` to be skipped, or `SKIPch621` which causes all tests in the `ch621` function to be skipped. You can record the reasons for each compile-time skipped case or run-time failure. In your flags.h file, you can add a definition to some compile-time flags, such as

```
#define SKIPch621 our parser error
#define FAILch622  Plum Hall bug?
```

Once you've categorized your skips and fails in this way, the strings you defined will show up in the execution output, something like this:

```
#SKIPPED: ch621 (>our parser error<)
#FAILED: ch622 (>Plum Hall bug?<)
```

The "unexpected" skips and fails will show up with the distinctive string "(><)" attached to each "unexpected" skip or fail. This makes it much easier to re-run the test suite after you've made compiler changes, because you can quickly search for the "(><)" string in the output to see if any new errors have appeared.

Running the CONFORM Programs

The previous Section described the configuration process. Once configuration is completed, you are ready to compile and execute the `CONFORM` programs. Any compile errors reported may represent currently-unimplemented syntactic features, or bugs in your compiler, or bugs in the Suite. Or, don't forget, sometimes a compile error means that the compiler wasn't properly installed, or that you weren't told the proper command-line options to use, or that the compilation environment wasn't properly set up. You have to investigate all these possibilities.

If you are unable to trace the cause of any compile errors whilst building CONFORM, you should telephone, fax or email Plum Hall for assistance.

File Naming Conventions

The file names and function names are all keyed to the section numbers of ANSI/ISO 9899:1999. This is simply achieved by concatenating the digits from the relevant section number to form a filename. Thus, file ch61.c deals with section 6.1 of the Standard.

Since identifiers (and, in some operating systems, file names) have to start with letters, a prefix is applied. The actual prefix used is defined by the nature of the file or function:

ch6 section 6 of the Standard;
ch7 section 7 of the Standard;
n files in CONFORM/NEGTESTS containing "mandatory" diagnostic situations—syntax or constraint errors;
q files in OPTIONAL/OPTAUTO containing "quality" diagnostic situations—undefined behaviors of one sort or another.

Bounds-Checking TR (now Annex K of C11)

The C standards committee (JTC 1/SC22/WG14, working closely with the US committee PL22.11) developed a Technical Report "Extensions to the C Library – Part 1: Bounds-checking interfaces", also known as the "Bounds-Checking TR". The TR is available through your channels for ISO standards as TR 24731-1, and a recent draft is found at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1225.pdf. All sections of the TR are now tested in CV-Suite in ch7*.c. To enable the new tests, put into your "flags.h" file a new definition for #define __STDC_WANT_LIB_EXT1__. These library functions have become an optional part of the C11 standard.

Unicode Strings TR (now part of C11)

The C standards committee (JTC 1/SC22/WG14, working closely with the US committee PL22.11) developed a Technical Report "Extensions for the programming language C to support new character data types", also known as the "Unicode Strings TR". The TR is available through your channels for ISO standards as TR 19769, and a recent draft is found at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1040.pdf. Some sections of the TR are now tested in CV-Suite in ch7_24f. To enable the new tests, put into your flags.h file a new definition for #define WANT_NEW_UNICODE_STRINGS. These features have become a required part of the C11 standard; they are also a required part of the C++11 standard .

Tracing

There is an extensive "trace" capability; Many of the routines in util.c receive a line-number argument (e.g., iequals). Each of these routines will, if the global variable Debug is non-zero, print a diagnostic trace to the standard-error output.

This trace keeps track of the source-file and line numbers of all statements reached.

The environ program will set Debug to non-zero if environ is invoked with two command-line arguments. (Its first command-line argument must be a "1".)

Each of the other executable programs in CONFORM—lang, prec1, prec2, and lib—will set Debug to non-zero if the program is invoked with any command-line argument. Thus, to execute the lib program with debug-tracing, you should execute lib debug .

Floating-point Comparisons

Up through 2010, if the user defined the FREESTANDING macro, then CV-Suite made no use of the compiler's library. However, we have encountered problems in the comparison of floating-point numbers when one or both of the values is NaN. There is now a macro IS_NAN(x) in flags.h which produces an invocation of the C99 library function isnan(x). This macro is invoked in several floating-point comparison functions in util.c, so that if an obtained result and the expected result are both NaN, the comparison is scored as a successful match.

If your target compiler does not support the C99 isnan(), then you must provide a different definition for IS_NAN(x). For example, you could #define IS_NAN(x) (!((x)!=(x)) && !((x)==(x)))

(depending upon your compiler's floating-point operations). If there are no NaNs in your target environment, you could define `IS_NAN(x)` as zero.

In `util.c`, when the `FREESTANDING` macro is not defined, the printing of comparison errors will make use of the target compiler's `snprintf` function. If the target compiler's library does not provide the `snprintf` function, you could modify `util.c` to perform the `FREESTANDING` logic instead; or you might `#define snprintf _sprintf_s` (or any other implementation-defined alternative; or you could modify `util.c` to use the `sprintf` function in place of `snprintf`.

When CV-Suite is being used to test conformance to an ISO standard, then the decisions of the ISO C committee are relevant to the required accuracy of floating-point comparisons; specifically, there is no mandate for any specific level of required accuracy. Over the past decades, the marketplace has generally accepted the criteria used in CV-Suite. When `double` or `long double` values are being compared, the comparison is by default performed at `float` resolution, with a permissible fuzz named `FDelta`. If the `double` or `long double` values would not compare equal in full precision, using the more restrictive fuzz named `Delta`, then a purely informational note is printed.

However, when CV-Suite is being used for internal QA purposes, you can compare `double` or `long double` values with their full precision, by modifying `util.c` to initialize `test_full_dequals_ldequals` to a nonzero value.

In making these changes, we have made several other changes to the floating-point comparison routines, but none those changes are intended to have user-visible consequences. As always, contact `support@plumhall.com` if you have corrections or suggestions.

Negative Tests in NEGTESTS

The Standard categorizes certain constructs as erroneous (or at least non-portable); in `CONFORM/NEGTESTS` (and in `OPTIONAL/OPTAUTO`) there are many little negative testcases for assessing the diagnostic messages of your implementation.

Here is an example from `conform/negtests/n01.in`:

```
/* #013 C90 6.1.2 C99 6.4.1         CONSTRAINT-MANDATORY  */
 /*   keywords are reserved (in phases 7+8 language syntax)   */
int main() {
      {char *break; }
      return 0; }
```

This refers to a specific subclause from the ANSI/ISO C standard, namely 6.1.2 in the C90 version and 6.4.1 in the C99 version. The category of the requirement is "CONSTRAINT-MANDATORY", and the underlying category of the standard is a "constraint". The testcase is the 13th testcase in the file, as indicated by "`#013`". The initial comment is followed by the contents of the testcase.

Note that the Standard allows diagnostics to be generated by any phase of the translator, including the linker. You may, therefore, need to perform a compile-and-link to produce a required diagnostic.

The files `n01.in` through `n10.in` are concatenations of the test cases for each Section.

The script file `section` (or `section.bat`) controls the automated execution of sections of the negative-tests. For example,

```
      section n01
```

splits `n01.in` into individual `.c` source-files (using the `unarc` tool), compiles each of these files to produce a `.clg` (for "compile log") message output file, then runs score to count how many diagnostic messages were and were not produced, producing a `.out` output file. Note that the input file `n01.in` contains testcases up to C99 subclause 6.4; `n05.in` contains testcases for C99 subclause 6.5, `n06.in` covers 6.6, etc., up through `n10.in` for subclause 6.10.

(The `score` program uses the `gotdiag.h` header that you configured when installing in order to determine just what counts as a "diagnostic message" in your environment.)

There is a kludge involved in `section`: in order to control the whole process using capabilities that are present in ordinary command languages (such as `shell` and `COMMAND.COM`), the score program creates, for each source-file

(e.g. n01001.c), another (empty) marker file (e.g. n01001) to satisfy a wild-card match loop line in the section script. These files are removed at the end of executing section.

Expected NEGTESTS

The summary utility program distinguishes "expected" failures from "unexpected" failures, but in order to do so, some user input is needed. For the positive tests, we have for many years provided the opportunity to put "FAIL" flags in "flags.h", but no corresponding method was available for negative tests. Now, for each NEGTEST component (such as n05) you can provide a list of expected failures in a ".exp" file (such as n05.exp), formatted as a sequence of three-digit case numbers, each terminated by a comma (including the final one). Thus if you know that your compiler will produce no diagnostics for cases n05003, n05008, and n05017, you could create a file in your destination NEGTESTS directory named n05.exp containing just one line: "003,008,017,". The score utility program will read this file, and on each "NO DIAGNOSTICS" message it will append the string "(>expected<)". The existing summary utility program thus categorizes these test cases as "expected" (i.e., not "unexpected") when it tabulates the test results.

Capacity Tests in CONFORM/CAPACITY

The file capacity.c meets all the translation-environment limits provided in C99. It should compile without errors, and produce the output message:

```
SUCCESSFUL EXECUTION
***** 1 successful test in CAPACITY *****
***** 0 errors found in CAPACITY *****
***** 0 remarks found in CAPACITY *****
***** 0 skipped sections in CAPACITY *****
```

Testing the C features of a C++ Compiler, in Suite++ CONFORM/CTESTS

We have started to provide a flags.h file, and a makefile, for testing the C behaviors of a C++ compiler. These tests will be built in the CONFORM/CTESTS subdirectory under your Suite++ destination directory. When you have located this subdirectory, copy (from your CV-Suite sources) the file dst-ix/ctflags.h to flags.h in that subdirectory, and copy dst-ix/ctmake to makefile, in that subdirectory (for Windows, copy from dst-win). We have configured the tests for cvs16a (and subsequent releases) to incorporate the C++ feature flags (CXX03, CXX11, CXX14); you must edit the flags.h file to choose your C++ standard.

Expression Tests in CONFORM/EXPRTEST

The EXPRTEST directory contains about one megabyte of selected outputs from the EXIN and EGEN components of the Suite. These are complete C source files, ready to compile and execute, which test the expression-code generator of your implementation. Further testcases of the same sort are provided in the EXPRTEST.95 and EXPRTEST.98 directories.

Summarizing the results from CONFORM

When you have tried all the components of the CONFORM section (or at any time you like), you can compare your obtained results against the expected results by executing, in the destination root directory, the command

```
make-summary
```

which will summarize the results against your choice of expected-results files.

The output gives a detailed list of numbers of tests expected, succeeded, failed, skipped, unexpected, and missing. The number of "unexpected" equals the total of failed and skipped, minus the number of "expected" errors, recognized by the presence of text between the "(>" and "<)" markers.

Checklist for CONFORM

Make a destination root directory
Verify sufficient free space on destination drive
Copy `dst-ix` or `dst-win` into destination root
Determine the target compiler, its flags, and its invocation
Determine the host compiler, its flags, and its invocation
Edit `envsuite` for appropriate flags
Edit `compiler`
Edit `linker`
Edit `hocompil`
Edit `holinker`
Edit `flags.h`
Edit other scripts? `execute, section, cleanup`
[If target not ANSI/ISO] configure `defs.h, compil.h, machin.h`
[If host not ANSI/ISO] configure `hodefs.h, hocompil.h, homachin.h`
Edit `gotdiag.h` to recognize target compiler's diagnostics
"Source" the `envsuite` (or `envsuite.bat`)
Type `set` to verify correct environment settings
Invoke `make all` to build tools programs
When `txtchk` is built, verify checksums back in source root
Run `buildmax` all to run all the CONFORM tests
Open a second window in the source directory, to look at sources
Edit `flags.h` with "skip flags" to "work-around" compile errors
Tabulate results, using `make-summary`
Deliver our results, and celebrate

EXP Files (Expected Results)

There are several `.exp` files associated with the test suite these are summaries of expected results, but all have specific uses as described below:

`conform-c90.exp` `conform-c90-fs.exp`	Expected results for conformance against ISO 9899 :1990 and ANSI C X3.169.1989, including ISO Amendment 1 and the Technical Corrigenda TC1, TC2, and TC3. (The "`-fs`" tests are "freestanding".)
`conform-c99.exp` `conform-c99-fs.exp`	Conformance tests against ISO 9899:1999. (The "`-fs`" tests are "freestanding".)
`conform-c11.exp` `conform-c11-fs.exp`	Conformance tests against ISO 9899:2011. (The "`-fs`" tests are "freestanding".)

UNIX considerations

If you are on a UNIX platform, you may need to execute the `chmodall` script:

```
sh chmodall
```

in order to mark all your script files as executable files. (It can't hurt, whether needed or not.)

When you edit these script files, note that the comment notation `###` indicates something that you may need to modify— compiler-specific logic in a script, or system-dependent name formats, etc.

DOS considerations

The scripts and makefiles need three commands which are common on UNIX but not standard on MS-DOS: **`cat`**, **`rm`** and **`cp`**. We have written work-alike C source files named **`phcat.c`** (for ``Plum Hall cat"), **`phcp.c`** (for ``Plum Hall cp"), and **`phrm.c`** (for "Plum Hall rm"). The **`makefile`** in **`dst-win`** will compile these to produce exe files (**`phcat.exe`**, **`phcp.exe`**, **`phrm.exe`**). After building each of these exe files, the **`makefile`** invokes a "setup" script (**`setup-cat.bat`**, **`setup-cp.bat`**, **`setup-rm.bat`**). Using "cat" as an example, the setup script determines whether a command named **`cat`** is already available on this system. If not, it copies **`phcat.exe`** to be named **`cat.exe`**, so that any further invocation of **`cat`** will invoke this exe file.

Makefiles

If you are using scripts (as described above) your `makefile`s will need very little customizing. Be sure to define the environment variable PHMAKE, to define the name (and arguments if desired) of your `make` utility, such as `nmake` , or `make -k` , etc.

All the file-name extensions are parameterized as macros taken from the external environment.

For example, the makefile rules governing object-files will use the `$(OBJ)` macro read from the external environment (which were put there by the `envsuite` script when you started working in this destination tree).

All of the specific actions in the distributed `makefile` are expressed with the scripts (as described above) `compiler`, `linker` etc.

Each directory of the Suite has its own `makefile`.

Many versions of make will accept a `-k` flag, which tells make to *keep trying*; if make encounters errors on one production, it goes to another one, so long as it does not depend upon a prior unsuccessful production. This is preferable to marking actions as optional (with leading hyphen) because productions that depend upon the result of a failed step should not be started.

Using BUILDMAX to Execute all Makefiles

When you have configured for your choices of environment, you should be ready to execute the `makefile` that you find in each directory.

After each individual `makefile` has been tested, the `buildmax` command can be used to iterate a command over the `makefiles` in each of the conform directories: (The `buildmax` command uses the `PH_STD` variable from `envsuite` to determine which standard should be tested.)

`buildmax all`	Rebuilds all makefiles.
`buildmax clean`	Cleans up everything but the `pgm.out` outputs
`buildmax clobber`	Removes even the `.out` and `.log` files

The same options are available for the `make-c90` command (for the pre-C99 tests), the `make-c99` command (for the C99 tests), and the `make-c11` command (for the C11 tests). But these commands require an explicit argument for the freestanding tests, e.g.

`make-c11 fs all`	Rebuilds all makefiles for FREESTANDING test

Separate Directory Trees

Keeping the intermediate files and results in a tree separate from the sources allows simpler configuration control.

As you work with the harness, there is always only one active *hosted* compiler, which you use for building tools, and one active *target* compiler, which you use for running the tests.

You choose the active compilers, and their associated environment variables, by executing envsuite.

Working on a multi-window system

If your environment allows you to open several different windows, you may find it easier if you use one window to work in the source tree, and use another window to work in the destination tree. See the earlier discussion in *An Example Session.*

Distribution

Directories In the C Suite (and in the C++ Suite), we provide two separate trees:

`dst-win`	a Microsoft Windows destination directory
`dst-ix`	UNIX destination directory,

each of which has a structure that matches the overall Suite.

So, to get started with testing a new compiler, make a directory or your own destination (e.g. `dst`), and do a recursive copy of the most suitable, `dst-win or dst-ix,` to your `dst` directory.

Configuring Files in the Destination Directory

Once installed, change directory to your `dst` directory, and hand-modify the script files that you find there (as described above).

Besides the scripts, you will need to configure these other files that are in your chosen `dst` directory:

`gotdiag.h`	specifies how to tell whether `pgm.clg` contains a "diagnostic message" (as required for the error-tests cases)
`hocompil.h`	characteristics of host-compiler (if different from target compiler)
`homachin.h`	characteristics of host-machine (if different from target machine)
`hodefs.h`	flags for hosted compilation (if different from `defs.h`)

Setting the ENVSUITE environment

Each time you change directory into a particular destination tree, it is very important to *source* the `envsuite` script of that particular tree to establish all the necessary environment variables. This operation exports the environment variables into your interactive shell.

You do this in different ways depending on your operating system:

- For DOS, you simply type `envsuite`
- For Bourne shell, use the dot command: `. ./envsuite`

Compiling the TOOLS Programs

When you make all in the top-level destination directory, you will build all the executable harness tools that are supplied with the Suite:

`unarc`	extract individual negative-test `.c` files from `.in` files
`score`	check for presence of diagnostic messages in `errauto` output
`summary`	tabulate all Suite outputs, compare with expected counts
`txtchk`	verify checksums for files

These same tools are also distributed with Suite++, the Plum Hall Validation Suite for C++. All of them are capable of being compiled as C; some of them must be compiled as C.

Producing a Summary of Results

The `summary` tool scores your results against an expected-results file (see "EXP Files" above). It is invoked in this fashion:

```
summary -f conform-c99.exp >conform-c99.sum
```

The resulting `conform-c99.sum` file might look something like this:

```
EXPECTED  ACTUAL  ERRORS SKIPPED UNEXPEC FILE NAME  Plum Hall CV-Suite 16a
       1 MISSING ....... ....... .......   conform/capacity/capacity.out
      20      20       0       0       0   conform/environ.out
      59      58       1       0       1 **conform/lang.out
    7742    7740       0       2       2 **conform/lib.out
    1642    1642       0       0       0   conform/prec1.out
    2413    2413       0       0       0   conform/prec2.out
      37       7      30       0       9 **conform/negtests/n01.out
  [...]
      21      21       0       0       0   interps/interp94.out
   46040   45739     292       9      18   TOTAL
```

The first column ("expected") gives the total number of test cases in that one output file. The second column ("actual") gives the number of test cases completed successfully. The third column ("errors") gives the number of test cases that failed. The fourth column ("skipped") gives the number of test cases that were skipped (using the `SKIP` flags in `"flags.h"`). The fifth column ("unexpected") gives the number of unexpected fails and skips. The sixth column ("file name") gives the name of the output file being scored. If that output file had unexpected fails or skips, the name is prefixed with two asterisks ("**"). If that output file was never produced, the "actual" entry is shown as `"MISSING";` see the entry for `capacity.out` in the example above.

As you see, the `summary` tool gives a quick overview of the test results, and shows which output files might need more detailed scrutiny to understand the problems revealed by the suite.

Timings and Sizes

In all these instructions, we assume that you have a system such as MS-DOS or UNIX which has a hierarchical directory structure. If not, you will need to alter the procedures whenever directories are discussed. This will greatly affect the time taken to install and configure the suite.

After installation you will have these directories:

Directory	**Description**
`../cvs16a`	the "home" Suite directory

`./doc`	documentation directory
`./bench`	sample benchmark programs
`./dst-win`	destination tree for some Win32 compiler
`./dst-ix`	destination tree for some UNIX compiler
`./conform`	all the official conformance tests
`exprtest`	tests the expression code-generator
`exprtest.95`	tests the expression code-generator
`exprtest.98`	tests the expression code generator
`negtests`	tests required diagnostics
`errauto`	“old” C90 diagnostics tests
`capacity`	verify compiler meets capacity requirements
`./interps`	tests "Defect Reports"
`./optional`	suggested but not mandated by the Standard
`./legacy`	Obsolete files, no longer used
`./testing/exin`	the Executive Interpreter
`./testing/egen`	the Expression GENerator
`./testing/limits`	probe capacity limits
`./testing/cover`	Generate self-checking expression tests
`./testing/stress`	endless expression-testing

Configuration Time

If you use the *script* approach to harnessing the Suite, it may take you from an hour to half a day to create your scripts the first time. (UNIX scripts are the easiest and quickest, other systems have more warts to work around.)

When you have properly configured your host-compiler, compiling the tool programs should take two or three minutes, tops.

If your compiler has good error recovery, and doesn’t ‘bomb out or hang-up’ when the Suite says nasty things to it, compiling the programs in `CONFORM` takes only several minutes.

Running the `CONFORM` sub-directories may take a few minutes more to extract, compile, and score all the programs.

An Example Session

Before describing all the details of configuring and running the Suite, we’ll take you through a hypothetical example session.

Let us suppose that we are working together in a laboratory that has two machines to be used in the testing, a file server *UNIX Server*, and a UNIX workstation *UNIX Box gamma* networked to the server.

Dual UNIX System Example Configuration

Installing and running the Suite might go something like this.

We `login` on the Server, and make a directory (`~/PlumHall/cvs22a`) where we want to store the original sources. (We’ll refer often to this directory; it is the *source root* directory.)

Then we extract the distribution into the source root directory; refer to instructions that came with the distribution.

```
[server]: cd ~/server/PlumHall/cvs22a
```

If we already have a copy of the Plum Hall checksum program on the server, we execute it now. Its output shows that the installed files agree with their expected checksums:

```
[server]: txtchk -f cvs22a
CVS22a_2016-03-31
Files are installed correctly.
```

If we don't already have an executable copy of the `txtchk` tool, we will soon be building one, and we can use it when we build it. (For Windows and MS-DOS systems—for the rest of this example just "DOS"—an already-built `txtchk.exe` is available in `dst-win` .)

Now that the sources are in the source root, and their contents are verified, we'll make them read-only so that we don't subsequently alter them during our testing.

```
[server]: pwd                  # be sure we're in source root
~/PlumHall/cvs22a              # yes, that's it
                               # now make files executable
[server]: cd dst-ix; sh chmodall; cd ..
                               # and non-writeable
[server]: find . -type f -exec chmod -w {} \;
```

(For DOS, use `dir` and `attrib`.)

Now we walk over to our first *target* machine, the UNIX system gamma, and `login` there. This gamma machine has a compiler called `acc` which we are supposed to test using the Plum Hall Validation Suite for C. Then we make a directory to work in.

This is our *destination root* directory, and we `cd` to that directory. (In recent years, we have settled upon a convention of naming the destination root directory as "CVSversion −CompilerName − StandardVersion", but any choice of destination root name is ok as long as you are consistent.).

```
[gamma]: mkdir ~/PlumHall/cvs22a-gcc-c20
[gamma]: mkdir ~/PlumHall/cvs22a-gcc-c20-setup
[gamma]: cd ~/PlumHall/cvs22a-gcc-c11
```

Next, we want to copy the entire `dst-ix` tree from the source root over to our new destination root:

```
[gamma]: cp -rp ~/PlumHall/cvs22a/dst-ix/* .
```

(For DOS, use `xcopy` with at least the `/s` and `/e` options.)

All the executable tool programs will be built in the destination root directory, and most of the configurable files (headers, scripts, etc.) also reside in the destination root.

Now we start configuring, so we can test the `acc` compiler. It's supposed to be a fairly robust, stable compiler, so we'll use it for our *host compiler* (to build our tools) as well as for our *target compiler* (to be tested).

There are three files that need to be edited, so we show them here as one recipe. We have drawn a box around this command; you will execute this command so frequently that it is worth memorizing. (On DOS, use your editor, perhaps incorporated in your compiler's IDE, or `edit.exe` or `notepad.exe`.)

`[UNIX]vi save-setup flags.h envsuite`
`[DOS] vi save-setup.bat flags.h envsuite.bat`

In the `save-setup` file, all we need to do is change the destination root to our own choice, and delete some line(s)

```
PHDST=~/PlumHall/cvs22a-gcc-c20
```

In the `flags.h` file, we un-comment the line corresponding to our choice of the target standard (`C90`, `C99`, or `C11`).

Next, we edit the environment setting file, `envsuite` (`envsuite.bat` in DOS), filling in the information required in order to run our `acc` compiler:

```
export PH_VSNAME=cvs22a
export PH_CCNAME=gcc
export PH_STD=c11

export PH_FREESTANDING=

export PHDST=~/Plumhall/cvs22a-gcc-c20
export PHSRC=~/PlumHall/cvs22a
  […]
export PHCFLAGS="-c -SPECIAL_acc_FLAG"
export CFLAGS=$PHCFLAGS
  […]
export PHCC=gcc
export PHHOCC=gcc
export PHCCONLY=-c

export PHLFLAGS=-lm
export LFLAGS=$PHLFLAGS
export OBJ=.o
export EXE=
  […]
export HOCFLAGS="-c -SPECIAL_acc_FLAG"
export HOLFLAGS=-lm
export OBJHO=.o
export EXEHO=
```

(How did we know what flags to use? See "What You Need to Know" above; someone familiar with `acc` had to tell us the proper flags for using `acc`.)

At this point, we need to run the `save-setup` script, so that we have saved the setup files.

```
[gamma]: sh -x save-setup
```

Next, we have to look at some script files named `compiler`, `linker`, `hocompil`, and `holinker`. (In general, the DOS "script" versions are batch files with `.bat` appended.) We're lucky; the scripts as delivered invoke the compiler using the environment variables `$PHCC` and `$PHHOCC` which we configured in our `envsuite` file. (You'll need to know how script files work, and how your target compiler is invoked, in order to know what changes need to be made to the script files as delivered.)

There are also some files to be copied into destination root from the source root, such as `conform-c90.exp`, `conform-c99.exp`, `conform-c11.exp`. There are the files of expected results that we will later compare our results against.

There are some other script files that sometimes need to be configured; see the Scripts section later in this chapter for details. But on most UNIX system platforms, the compiler and linker scripts are the only ones you might need to modify.

There are some header files in the destination root that might need to be configured. But we're told that the `acc` compiler is a fully ANSI/ISO compliant compiler, so we don't need to configure anything in the `compil.h`, or `machin.h` header files.

There is one header that always has to be checked. It is called `gotdiag.h` (got a diagnostic?), and it is used in the score tool that scores whether the target compiler produces diagnostic messages for all the deliberately erroneous *negative-test* files.

We need to find out how one can tell, looking at the output of our `acc` compiler, whether a diagnostic message was produced. It's not enough just to ask whether the compiler-output file is non-empty; the compiler always produces a copyright banner `© Copyright GAMMA Hypothetical Software, 2016`.

At this point, we have to discuss this with our company's compiler experts. We find out that the only way to tell that `acc` produced a diagnostic is to look for the characters `"error"` or `"warning"` in the compiler output, and the

header gotdiag.h already searches for these words. (As delivered, the gotdiag.h header is usually adequate, but it is important to check.)

If we previously required further customizations, and we saved those changes in our "setup" folder (typically named ~/PlumHall/cvs22a-gcc-c20-setup), then this is the time that we should restore those customizations (typically by copying some files from ~/PlumHall/cvs22a-gcc-c20 to ~/PlumHall/cvs22a-gcc-c20-setup).

Now we're ready to build the tools programs. Before we start any testing session, we have to source the environment variables in the envsuite script.

We're using the Bourne shell, so we type

```
[gamma]: . ./envsuite
```

(On DOS, we execute envsuite.bat.)

Now that we have executed the envsuite script (which on Unix/Linux marks commands as "executable"), we can save the files we have modified (flags.h and envsuite) by simply typing:

```
[gamma]: save-setup
```

Just for reassurance, we type set to look at the environment settings. (The set command works similarly in DOS.)

```
[gamma]: set
CFLAGS=-c -SPECIAL_acc_FLAG
EXE=
EXEHO=
HOCFLAGS=-c -SPECIAL_acc_FLAG
HOLFLAGS=-lm
LFLAGS=-lm
OBJ=.o
OBJHO=.o
PATH=.:~/PlumHall/cvs22a-gcc-c20:/bin:/usr/bin
PHDST=~/PlumHall/cvs22a-gcc-c20
PHSRC=~/PlumHall/cvs22a
```

This looks good, so we're ready to build the tools. We invoke make, and make prints out the commands it's executing. (For DOS, the compiler environment would determine how make is used.)

```
[gamma]: make all
hocompil score ~/PlumHall/cvs22a ~/PlumHall/cvs22a-gcc-c20
gcc: program not found
make terminated
```

Whoops, some kind of problem has surfaced. The make program says it can't find the acc program. We do some quick research, and discover that we have to add another directory to our PATH variable. So we once again edit the envsuite script, adding /gamma/special-bin to the initialization of PATH. Then we have to once again *source* the envsuite script (and save the modified script):

```
[gamma]: . ./envsuite
[gamma]: save-setup
```

Now we once again run make all, and everything runs successfully. All our tool programs are built, including the "textfile-checksum" program named txtchk. Just for good measure, we're going to change directory over to the source root, and repeat the txtchk checks there, as we did a few pages ago.

With the tools programs all built, we're ready to start the real testing. We will start by changing directory to the conform directory, and building one of the test programs that almost-always works correctly the first time, the first "precedence" program, prec1.c, which we pronounce here as "press one"):

```
[gamma]: cd conform
[gamma]: make prec1.o
```

Often, the very first compile terminates quickly, with an error message complaining about not being able to find some header file. Usually that kind of problem is fixed by adding all the necessary header-file directories to the "compiler flags" environment variable `PHCFLAGS`. When compilation is successfully producing `prec1.o`, we can go on to linking and executing:

```
[gamma]: make prec1.out
```

Then we can compile, link, and execute everything in the `conform` directory:

```
[gamma]: make all
```

Now come a few dozen lines of output from make. Suddenly it stops. A compile error has terminated the compilation of one of the files, the file `ch6_10.c` in the conform directory. We want to edit the compiler output log for `ch6_10`; it's named `ch6_10.clg`

```
[gamma]: vi ch6_10.clg
DIAGNOSTIC: ch6_10.c, line 421: undefined __STDC__
```

This sounds like a problem to be investigated; the special macro name `__STDC__` is supposed to be predefined in any ANSI/ISO C compiler.

Meanwhile, we have to find a *work-around* so that we can continue.

Let's go look at the source code inside `ch6_10.c`. We'll keep one window in the destination tree, and the other window over in the source tree. Here's what we do in the *source* window:

```
[gamma]: cd /gamma/mine/dst-ix # start at destination root
[gamma]: . envsuite     # source the environment settings
[gamma]: cd /server/PlumHall/cvs16a
                       # go over to source root on server
[gamma]: cd conform          # cd to conform subdirectory
[gamma]: vi ch6_10.c         # edit the original source file
```

What we find in `ch6_10.c` is that we can "skip" the contents of this file if we can `#define` a flag called SKIPCH6_10. Back over in the destination root there is a header file called `flags.h`. Any `#define` s that we edit into "flags.h" will be `#include`'d into each compilation

In our *destination* window:

```
[gamma]: vi ../flags.h # edit the compile-time flags file
#define SKIPCH6_10 1
:wq
[gamma]: save-setup
[gamma]: buildmax
```

Let's suppose that we are now so lucky that everything else runs smoothly to completion. We can obtain a summary report of all our results, using the `summary` tool and the `conform-c11.exp` (or `conform-c99.exp,` or `conform-c90.exp)` file of expected results, in the *destination* window:

```
[gamma]: make-summary
[gamma]: cat conform-c11.sum
EXPECTED ACTUAL ERRORS SKIPPED FILE NAME  CV-Suite 16a 2016-03-31
       1      1      0       0    conform/capacity/capacity.out
      20     20      0       0    conform/environ.out
     ...    ...
      21     21      0       0    interps/interp94.out
   46040  46029     10       1    TOTAL
```

From this, we see that there are 10 run-time errors, and one skipped section (presumably the `ch6_10` section that we skipped). We will deliver this `conform-c11.sum` summary, along with the detailed `.out` output files, to the compiler group which is responsible for interpreting the results. We, for our efforts, deserve a lunch break.

Files to be Configured

There are several header files that may need to be configured:

`compil.h`	specifies syntactic and semantic features of the compiler

machin.h	specifies machine-dependent characteristics, such as the range of integer data types
defs.h	defines useful macros, as well as #including both compil.h and machin.h
makefile	specifies the rules for building the components of this directory

Once defs.h (with compil.h and machin.h) have been successfully configured, they will be used by all components of the Suite, both tools and tests. Each directory has a makefile that might require adjustment to suit your environment.

Configuring defs.h

The files compil.h and machin.h define several preprocessor variables with values that are appropriate to strict ANSI C. Among other things, they define the macro name ANSI to 1. In a full ANSI environment, no further customization should be needed or allowed, since the standard headers <limits.h> and <float.h> specify all that is needed about the nature of the machine.

If you are testing compilers that are not strict ANSI/ISO C, look at the further configuration details in the Section on Configuring Headers.

Testing a Freestanding Environment

The Suite components are structured so that a freestanding environment can be tested with a minimum of special configuration.

You should add this line to defs.h and flags.h:

```
#define FREESTANDING 1
```

In the destination root directory, you should configure the file sdutil.h. This file contains the only remaining hosted environment assumptions about the availability of file I/O. Revise it to accommodate the method you use to get output from your in-circuit emulator, simulator, or whatever.

In the header sdutil.h, the function pr_ok takes one argument, a char string named s that must be sent to the external environment. In a hosted environment with files, the argument s is simply sent to the standard output using fputs; but in an embedded environment, use the appropriate interface of the simulator or the testbed.

The function setzero exists in order to prevent the compiler's optimizer from recognizing that the global integer variable named Zero is, in fact, always equal to zero. The version of setzero provided in sdutil.h attempts to fopen a file (which is fact does not exist) and to read a value for Zero. The simplest replacement for setzero is a simple empty stub. Whether anything further is needed will depend upon the goals for the testing, and the requirements for certification, if any.

Scripts

Using scripts or batch files for compiler, linker, etc., simplifies many aspects of executing the suite in varying environments. For example, many QA departments need to routinely re-execute the Suite using dozens of different compiler flags and options.

Using an unchanging set of compiler scripts, and just changing the flags and options in one script, or just setting the flags into environment variables, allows routine re-running of the Suite.

Here are the scripts that you may need to modify:

`compiler pgm src-dir [inc-dir... ]`
: Compile pgm, found in src-dir; headers also from inc-dir. Put output, especially diagnostic messages, into pgm.clg

`hocompil pgm src-dir [inc-dir... ]`
: Host-compile pgm, with src-dir and inc-dir as above. Put output, especially diagnostic messages, into pgm.clg

`linker main-obj [lib-obj... ]`
: Link object-modules to produce executable main program. Put output, especially diagnostic messages, into `pgm.llg`

`holinker main-obj [lib-obj... ]`
: Host-link object-modules to produce executable main program. Put output, especially diagnostic messages, into `pgm.llg`

`execute pgm [arg... ]`
: Execute target program `pgm`, capturing standard-output in `pgm.out` (When cross-compiling, this may involve networking executable `pgm` to development system, remote execution, and networking results back into `pgm.out`.)

`cleanup`
: Remove object files, executable files, intermediate files, etc.

`section archive src-dir`
: In negative-tests directory, `unarchive` and test one section of negative tests; score the results into output `archive.out`.

`make-summary`
: Using the appropriate file of expected results, produce the `.sum` file.

`envsuite`
: Refer to the `envsuite` script for detailed documentation of the environment variables to be defined for host and target compilers.

The TESTING Component

To this point, this user manual has described the conformance-test components of CV-Suite, which are used by various parties to determine conformance to the ANSI/ISO C standard. But CV-Suite also provides several components which are used for QA testing of C compilers, over and above the conformance requirements. These components are described below.

The BENCH benchmark tests

In the `bench` directory are source code for several small benchmark tests. The "destination" directories (`dst-win, dst-ix`) contain a `bench` directory. Each contains a `makefile` that will build the `bench` executable program, then execute that program to produce a `bench.out` output file.

The tests produce one line of output (at the end of many lines of trace information). If you combine each line of output into a table, you get a table like this one (bench-2001.tbl):

```
Example outputs circa 2001
Times are in nanoseconds
```

	register int	auto short	auto long	function call+ret	auto double
800mhz-pc	3.09	9.56	7.79	13.1	10.6
b55-presario-1600	6.85	8.32	7.89	12.6	23.0
b55-vaio-pcg-f540k	6.92	7.78	6.85	12.6	16.9
gcc2.96-1.2ghz-pc	3.9	3.49	3.24	6.75	3.85
msvc7-dell-inspir8000	4.77	4.86	4.74	6.26	3.46

More details about the benchmarks are provided as an appendix at the end of this user manual. Note that the older benchmark timings (1980's and early 1990's) are given in microseconds, not nanoseconds.

The OPTIONAL negative-tests

In the `optional` directory, there is a directory containing several negative-tests (tests for production of diagnostic messages). All of these tests invoke "undefined behavior", so a high-quality compiler should at least produce some form of "warning" message, but the C standard does not impose any requirements. The structure, and the harness, are the same as the `negtests` directory, as described above.

EXIN

`EXIN` is the Executive INterpreter. It is a script processing language, and is used for many of the more advanced tests in the Suite. The language processed by `EXIN` is inspired by `sh` and `csh` from the UNIX operating system.

The EXIN Command Line

The `EXIN` interpreter takes the following command line:

```
exin [-D] [-R] <filename> [<arguments>]
```

`EXIN` takes its input from the script filename specified on the command line, and processes one line of input at a time. The syntax is similar to the C language. There is one data type, a text string, but that can be evaluated numerically by built-in operators.

There is high level control flow (including `for` loops, `while` loops, `if` blocks and `switches`). Other programs can be executed (such as the compiler under test). `EXIN` can write text to files, and is commonly used to generate C programs.

The optional `-D` parameter specifies "debug" trace output. If this flag is specified, `EXIN` will leave an execution trace in a file named `exindebug`. This same trace behavior can be obtained by defining an external environment variable `EXINDEBUG`, which will cause debug output from every execution of `EXIN` (including nested invocations).

The optional `-R` parameter is a number that can be used to re-seed the random number generator.

Running EXIN

Once a compiler has passed the `CONFORM` section of the Suite, it can be assumed that compiler handles all of the syntax and semantics of the C language. The next step is to build `EXIN` and have it pass its own test suite.

EXIN is an extremely portable program. It can be compiled by a K&R, System V, V7, or ANSI implementation. However, if the implementation lacks a "spawn" capability (invoke a command and return its status code to the caller), it can only be used as a "generator" of output files. If a "spawn" capability is available (via the system function, or equivalent), EXIN can be used as a full command interpreter, to create test files, compile, link, and execute them. This extra capability is well worth the effort of configuring EXIN for "spawn" capability.

Before describing the syntax of EXIN scripts, we will describe how a script is executed:

1. The file is read into memory.
2. Data structures are created that describe all of the control structures; loops, switches, if statements are noted and information on their entries and exits is noted.
3. Each line is executed in order unless there is an explicit control flow construct.
4. As each line is executed, it goes through the following:
5. discard any text following # (i.e., a comment);
6. expand all variable (macro) references in the line;
7. do any I/O redirection;
8. parse the command;
9. if the line is an EXIN command, do the appropriate command;
10. otherwise attempt to pass it to the operating system.

EXIN Keywords

The EXIN interpreter uses the following list of keywords:

`and`	secondary control list for `for` loops
`break`	unconditional transfer out of a loop
`by`	numerical increment in a `for` loop
`case`	a value to be matched in a `switch`
`continue`	skip to the next iteration of a loop
`default`	if no cases in a `switch` match
`echo`	write rest of line to `STDOUT`
`echoerr`	write rest of line to `STDERR`
`else`	alternate control for if statement
`end`	marks end of all control structures
`exit`	terminate the script
`for`	iteration control
`goto`	unconditional control transfer to a label
`if`	conditional execution
`in`	keyword used in one variety of `for` loop
`set`	set the value of a local variable
`setenv`	set a global (environment) variable

`shift`	move command line arguments
`source`	execute a script in the current context
`switch`	control transfer selection
`to`	keyword used in one variety of `for` loop
`unset`	remove a local variable definition
`unsetenv`	remove a global variable definition
`while`	loop control

`EXIN` has control structures for `while` loops, `switch` statements, `if-else` statements, and both numerical and list directed for loops. All control structures terminate with the `end` statement, and can be nested to arbitrary depth.

FOR Loops

There are two forms and one variant of `for` loop structures. String loops iterate over items in a list. Each time through the loop the control variable is set to the next item of the list, until the list is exhausted:

```
for <name> in <list>
...
end
```

Numeric loops iterate over a numerical range. The control variable is incremented or decremented each time through the loop until the end value is reached:

```
for <index> = <start> to <end> [ by <incr> ]
...
end
```

The default value of `incr` is `1`, and loops always execute at least once.

The and keyword allows loops to have multiple parallel control variables:

```
for <name> in <name_list>
and <index> in <element_list>
and <index> = <lwb> to <upb> [by <incr>]
...
end
```

An `and` list or range must contain the same number of elements as its companion `for`, and all `and` statements must be on lines immediately following for statements. The `and`s terminate at the end of their companion for and do not require separate `end`s.

Textual and numerical loops may be intermixed in `for` / `and` groups, but the number of control variables must all come out even at loop termination.

WHILE Loops

The `while` loop iterates as long as test is true, i.e., as long as `$eval(<expression>)` is non-zero:

```
while <test>
...
end
```

BREAK and CONTINUE Statements

The `continue` statement transfers control to the `end` statement of nearest enclosing `for` or `while` loop, for its next iteration.

The `break` statement transfers control to the first line beyond the end of the nearest enclosing `for`, `while` or `switch` structure.

GOTO Statements

The `goto` statement transfers control to the first line after a label:

```
goto <label>
```

Transfers into block structures are not allowed but transfer out of a block structure is. A label is established with a colon as

```
:label
```

IF Statements

The `if` statement executes the body of the clause if the expression evaluates to a non-zero integer. The `else` clause is optional. The expression is always evaluated as if it were written:

```
if ( $eval ( <expression> ) )
...
[ else
... ]
end
```

SWITCH Statements

The `switch` statement tries to match the word to the arguments of the associated `case` statements, just like in C. Similarly, there is a `default` statement. If there is not a `break` at the end of the case, control flow will fall through to the next case.

```
switch <word>
   case <pattern>
...
   case <pattern>
...
   break
   default
...
end
```

The matching is a textual match, and certain wild card characters are allowed in the `case` arguments. A pattern is composed of a word or set of meta-characters:

`*` matches any 0 or more characters
`?` matches any single character
`[...]` matches any character in sets (`[a c d]`) and/or ranges (`[A-L]`) inside brackets

EXIT Statements

There are two forms of `exit`: with and without a numerical argument. If an argument is present, it is returned to the calling process. If not, a `0` is returned. If a file is being sourced, one file level is popped. Otherwise the current shell is exited. The error variable is set appropriately whether `exit` is from a source or the shell.

Variables

In addition to keywords, the `EXIN` language uses variables. All text that is not used as a keyword is treated as simply text, unless it is prefixed with the symbol `$`. This declares that a variable is being used, and variable name is to be substituted into the text.

For example if the variable `THIS` is currently defined to be `"this is it"`, then the line

```
echo $THIS THIS $THIS---
```

becomes

```
echo this is it THIS this is it---
```

Access to Command Line Arguments

EXIN supports named variables (described below), and variables defined on the command line.

$0	substitute name of current file
$n	substitute nth command line argument
$#	substitute the number of remaining arguments
$*	substitute the entire argument list
shift	rename $n etc. to be $n-1 etc.
shift 3	equivalent to shift; shift; shift

Local and Global Variables

The only data type in an EXIN script is a string of text. Variables may be created whose value is the currently assigned string. Whatever the size of a string, it is treated lexically as one "word". There are two kinds of variables:

set variables	local, limited to the lifetime of their shell
setenv variables	global, and inherited by any sub-shell

The syntax of the set command is as follows:

set <name> = <word>	defines <name> to be <word> string
set <name>	displays current definition
unset <name>	removes definition

Similarly for setenv and unsetenv.

Referencing Variables

When the $ character is seen, EXIN tries to make the longest possible match of a local or global variable. For example, if both ABC and ABCD are defined, $ABCD will be replaced with the current value of ABCD. If a name is defined both locally and globally, the local definition will be used.

$<name>	substitute contents of <name>
$$<name>	substitute for <name>, then rescan

If no definition for a name such as $ABCD is found, EXIN inquires from the external environment whether the name is defined there. That is, EXIN asks whether getenv("ABCD") finds a definition, and if so, EXIN copies that definition into its internal name table.

Referencing Inside Strings

It is possible to get the component parts of a variable using a C-like array syntax. Although each variable is treated as a single "word" during substitution, the individual component "words" are subscripted 1...n (unlike C language arrays, which are subscripted 0...n-1).

$<name>[n n ...]	nth components
$<name>[n-n]	components in specified range
$<name>#	number of components

For example,

```
set A = "one two three four"
echo $A[2 1] -- $A[3-4] -- $A#
```

produces

```
two one -- three four -- 4
```

Note: any of these indexes can be the result of an expanded expression. Thus,

```
echo $A[$A#]
```

produces four Built-in Variables

In addition to user-defined variables, there are some that are built-in to EXIN:

`$error`	contains return value of a sub-process
`$rand`	generates a signed integer random number
`$defined(var)`	produces 1 or 0 depending on whether

the string is the name of a variable

Expansion of Quotes and Escapes

Double quotes (`"`): All text enclosed between an open-quote and a close-quote is one logical word. Any variables in the string are expanded.

Single quotes (`'`): All text enclosed between an open-quote and a close-quote is one logical word. Variables are not substituted.

The single quotes can be used for dynamic binding:

```
set A = '$x' # line 1
set B = "$y" # line 2
...
echo $B $$A # line N
```

The expansion of `$B` contains whatever the value of `$y` was at line 2.

`$$A`, however, is expanded with whatever the value of `$x` was at line N.

Escapes (`\`): Escapes can be used to continue a logical line across a physical new line or to nullify or postpone the effects of special characters.

Expression Evaluation

The `$eval` operator expands all variables in its argument, then evaluates the string as an arithmetic expression. All of the operators of C are available with their natural precedence. Parentheses may be used to override the natural precedence. The numerical result is equivalent to 32 bit integer arithmetic evaluation of constant expressions in C.

Example:

`$eval($a + 1)` increments `$a`

All arithmetic operators of C language are recognized, as well as string comparisons and exponentiation:

`! ~ -`	Unary
`**`	Exponentiation
`/ %`	Multiplicative
`+ -`	Additive
`<< >>`	Shift
`< <= > >=`	Relational
`!= ==`	comparison (for strings, too)
`& ^ |`	Bitwise
`&& ||`	Logical

Input and Output Redirection

`EXIN` supports certain redirection facilities:

`<`	standard in
`>`	standard out
`>&`	standard out and standard error
`>o&`	standard out

`>e&`	standard error
`>oe&`	(also eo)

Examples:

```
cmd1 <file1 >file2
```

takes `stdin` from `file1, and` sends `stdout` to `file2`

```
cmd2 >oe&file3
```

send both `stdin` and `stderr` to `file3`

In addition, the >> symbol in place of > in the table will append the output to the named file.

Writing Output

The `echo` command writes the rest of the line to standard out:

```
echo This is what gets printed.
```

The `echoerr` command writes the rest of the line to standard error.

```
echoerr This gets printed. >& errfile
```

In this example, the string "`This gets printed`." has been redirected to the file `errfile`.

Executing Sub-Scripts

The `source` command allows other scripts to be executed in the current context. Such scripts have all of the local variables visible, and any definitions made in the sub-script will be visible to the parent. After `source newscript.ex` is executed, control goes to the next line beyond the source statement.

Sub-Programs

If a command is not recognized by `EXIN`, it is treated as an external sub-program. `EXIN` will attempt to "spawn" the sub-program and retrieve its return code (unless configured for "generate only").

Configuring EXIN with machdep.c

`EXIN` contains some machine and operating system dependencies that must be dealt with in order to build it. These dependencies have been isolated into the file `machdep.c`. For most UNIX-like systems, defining the preprocessor variable `fooNIX` will include the proper parts.

For Berkeley UNIX, the preprocessor symbol `BSD` should be defined.

For MS-DOS systems, the symbol `MSDOS` should work.

Unfortunately, this section of EXIN cannot be written in "portable" C. There are several reasons:

1. Spawning sub-processes: `EXIN` requires the ability to "spawn" a sub-process. Although the system routine is a "portable" way to do this, some versions of that routine do not return an error code from the child process. Using `spawn` (MS-DOS) or `fork/exec` (UNIX, POSIX) allows access to this return code (to set the `$error` variable). If your system is not supported already, we will help you in the porting process.
2. Re-directing input/output: The `open, seek` and `dup` calls are not supported in the ANSI Standard. It is not possible to redirect the I/O of a spawned sub-process without them. The `exin.h` file defines a symbol `NO_LEVEL_0`. If set to non-zero, only level 1 (`fopen` etc.) I/O will be performed. This allows all output from `EXIN` scripts to be redirected, but will not redirect the output of a sub-program. Again, if your system is not supported, we will help you through the port. See the discussion of the `GENERATE` option in the `LIMITS`, `COVER`, and `STRESS` sections of this reference.

There is a `makefile` for building `EXIN`. As with the other sections of the Suite, executing the command

```
make all
```

will create the `EXIN` executable, plus a small test.

Configuring EXIN: config.ex

EXIN is mostly used for controlling the building, compiling, linking, and executing of test programs.

The recipes that it follows for these operations are specified as options in the file config.ex, in the following strings:

COMPILE	command(s) for compiling the source file
COMPILE_OK	command for checking that the compilation succeeded
LINK	command(s) for linking the objects
CLEANUP	command(s) for deleting the source and objects; if DO_CLEANUP is set to NO then this is unnecessary
LOGFILE	file name in which to log results
GENERATE	NO if the files are to be generated compiled linked and executed
DO_CLEANUP	set to YES or NO to determine whether the generated files get cleaned up each time.
IF_ANSI	set to YES or NO depending on whether your compiler supports ANSI C (long double etc.).
OBJ	file extension for object files
EXE	file extension for executable files

The GENERATE Option

By default, the scripts in the COVER directory will generate, compile, link, and execute the test files. By setting the configuration variable GENERATE to be YES, the test files will be generated, but not compiled, linked, or executed. This is useful when you want to generate the files on one system and execute them on another. The standard scripts in the COVER section will generate almost 300 Megabytes of test files.

Testing EXIN

There are several EXIN scripts available for testing. Tests of the behavior of EXIN itself are obtained by executing

```
exin testall.ex
```

(Executing make all will perform this test also.) The results are self-explanatory: a test is run which produces output to the console. This is followed by the expected results.

Tests that exercise EXIN's ability to create, compile, link, and execute test programs are obtained by executing

```
exin comptest.ex
```

This test makes use of the config.ex file that was discussed earlier under Configuration. If any of the strings in config.ex were incorrectly configured, the error would be revealed as an error during the execution of exin comptest.ex.

EGEN

EGEN is the Expression GENerator. Since it is impossible to test all possible C language expressions, the Suite provides this tool for generating complex expressions, and code to check that the right answer is calculated.

The EGEN Command Line

`EGEN` is invoked with the following command syntax:

```
egen -D<data_set> [<flags>... ] [<template<]
```

The option flags are:

`R<number>`	random number seed
`S<name>`	subroutine name
`N<number>`	number of statements to generate
`I<name>`	file name to read input from
`O<name>`	file to write output to
`C`	emit check code at each statement (default)
`X`	suppress checking code until end of subroutine
`V`	use value-preserving (instead of unsigned-preserving) typing rules
`U`	use unsigned-preserving (instead of value-preserving) rules
`A`	use ANSI typing rules: shift takes type of left-hand-side
`K`	use K&R typing rules: shift follows usual arithmetic rules
`P`	generate only strictly-portable programs

There are no default typing rules; use either `-V -A` or `-U -K` .

The `EGEN data_set` is a text file which describes the variables to be used in generating the expressions. Several data sets are provided with the `STRESS` section, and others can be created as needed.

The `template` is a list of operators or special tokens that specify the kind of expression to be generated. Each token has an equivalent alphabetic name which can be used in its place (to avoid the need for quotes, backslashes, etc., in script files or `makefiles`).

Here are the `EGEN` operators and special token symbols:

`! not ~ compl`	Unary
`* times / div % rem`	Multiplicative
`+ plus - minus`	Additive
`<< lsh >> rsh`	Shift
`< lt > gt <= le >= ge`	Relational
`== eq != ne`	Comparison

`& band | bor ^ xor`	Bitwise
`&& andif || orelse`	Logical
`- neg`	unary minus
`pre++ preinc`	pre-increment
`pre-- predec`	pre-decrement
`post++ postinc`	post-increment
`post-- postdec`	post-decrement
`*= timeseq /= diveq`	assignment operators
`%= remeq`	" "
`+= pluseq -= minuseq`	" "
`<<= lsheq >>= rsheq`	" "
`&= andeq |= oreq`	" "
`^= xoreq`	" "
`= assign`	Assignment
`( lparen ) rparen`	parenthesis for grouping
`@ at`	EGEN randomly selects an operator
`{list}`	EGEN randomly selects an operator from the list
`lbrace list rbrace`	EGEN randomly selects an operator from the list

The expression templates can contain all C language unary or binary operators, as well as the special operators `@@` , `()` , and `{}` .

Ordinarily, each generated statement will be followed by code to check that the expression produces the correct answer, and that all side effects have taken place correctly. If the `-X` flag is specified on the command line, no checking code is emitted until the end of the module. This is useful for many compilers that lose common sub-expressions at function boundaries. More complete control of the statement and check process is available using the I input file option.

Running EGEN

After passing the previous sections of the Suite, a compiler should be trustworthy in calculating the results of simple expressions. `EGEN` relies on this to generate self-checking expressions of arbitrary complexity. Each complex expression has its value calculated from the simpler components that make it up.

For example, a compiler generating code for the statement

```
(a*b) + (c*d)
```

might have an error in keeping track of multiple registers and get the wrong answer. But calculated as

```
temp1 = a*b
temp2 = c*d
temp1+temp2
```

the right answer is more likely, given that expressions of this complexity have been exhaustively tested in the `COVER` section. This is the main idea of `EGEN`. By decomposing a complex expression into simpler pieces, `EGEN`

expects to get the "right" answer and use that to check the compiler's result on the full complex expression. In addition to the self-checking expression, EGEN puts comments into the generated source file that show the values of the simpler intermediate calculations.

An example of an EGEN command line is

```
egen -R23 -Dinteger.gen -N10 "(+)" "*" "(-)"
```

This sets the random number seed to 23, uses the data set defined in the file `integer.gen`, and generates 10 self-checking statements of the form

```
(variable + variable) * (variable - variable)
```

EGEN randomly assigns variables from the data set to each `variable`, and tracks what the final value should be. Given the command line

```
egen -Dinteger.gen -10 "{" "+=" "-=" "*=" "}" @
```

or the equivalent form with alphabetic names,

```
egen -Dinteger.gen -10 lbrace pluseq minuseq timeseq rbrace at
```

EGEN would generate 10 statements of the form

```
variable <OP1> variable <OP2> variable
```

where each variable is randomly chosen from the data set `integer`, `OP1` is randomly chosen from the set `{+=-=` and `OP2` is randomly chosen as any C operator. EGEN generates code for the expression, code to check the result of the expression, and code to check the results of any side-effects.

Defining an EGEN Data Set

The data set specified on the EGEN command line must contain a set of descriptions of C language variables. The format is identical to C variable declarations, but with a few limitations. The syntax supports all scalar types and any level of indirection. Floating-point initializers must have digits preceding the (optional) decimal point. All variables must be initialized, and naturally there are declaration order dependencies for pointers and the variables they are initialized to point to. Variables with storage class static can be initialized to point to local variables (which is neither legal nor meaningful in a C program). This latitude is available because all initialization code will be generated by run-time assignments.

For example:

```
auto int i = 3;
static int *pi = &i;
```

Here is a real example. This command line is

```
egen -Dinteger.gen -N4 "{" += -= "}" "{" neg ~ "}" "(" @ ")"
```

and the output is as follows:

```
main()
{
extern char *Filename;
int true = 1, false = 0;
auto unsigned int ui;
static unsigned int *pui;
auto int i;
static int *pi;
auto short s;
static short *ps;
auto char c;
static char *pc;
auto unsigned long ul;
static unsigned long *pul;
auto long l;
static long *pl;
register int rint1;
register int rint2;
ui = 3;
pui = &ui;
i = 10;
pi = &i;
s = 13;
ps = &s;
```

```
c = 20;
pc = &c;
ul = 65000;
pul = &ul;
l = 130000;
pl = &l;
rint1 = 1;
rint2 = 2;
Filename = "main";
iequals(__LINE__, rint2 -= - (*pui < c), 3);
iequals(__LINE__, rint2, 3);
iequals(__LINE__, *pi += - (s >>= ui), 9);
iequals(__LINE__, *pi, 9);
iequals(__LINE__, s, 1);
iequals(__LINE__, rint1 += ~ (true ? *pc : *ps), -20); iequals(__LINE__, rint1, -20);
lequals(__LINE__, *pl -= - (*pc /= *pui), 130006L);
lequals(__LINE__, *pl, 130006L);
iequals(__LINE__, *pc, 6);
report(__FILE__);
}
```

EGEN Input Files

If the `-I<filename>` option is indicated on the `EGEN` command line, then the generation process can be controlled from an input script. The contents of the file are copied directly to the `EGEN` output unless one of the `EGEN` keywords is seen:

`@header`	Generate the header code. This includes the subroutine entry point and the data declarations.
`@init`	Generate the initialization code.
`@statement`	Generate the statement. However many statements have been specified with the `-N<number>` switch will be generated here. If the `-X` switch is active, no checking code will be generated. This directive also takes an optional argument, a string, which can contain the same switches the command line accepts, with the exception of `-D`, `-S`, `-O` and `-I`. See the example below.
`@reset`	This resets the internal variables in `EGEN` so that an identical sequence can be generated. This is useful for creating instances of common sub-expressions.
`@check`	If the `-X` flag is active, the checking code is generated.

Here is an example of an input file:

```
@header

int my_var;
int my_array[10];
@init

/* move out loop invariants */
for (my_var = 0; my_var < 10; ++my_var)
   {
   my_array[my_var] = @statement "-X -N4 = { + - * / }";
   }
@check;

/* create a few common sub expressions */
```

```
@reset
@statement "-X -N4 = { + - * / }";
@reset
@statement "-X -N4 = { + - * / }";
@check
```

This example creates statements that are invariant with respect to the user-defined loop control variable. Then it creates two instances of the same four statements, which should produce common sub-expressions.

Statements that are not invariant with respect to user-defined loop control variables will not be handled properly by the current version of EGEN. (Please contact Plum Hall if you have suggestions for enhancement of EGEN in this area.)

EGEN64

EGEN64 is an enhanced version of the Expression GENerator. This version of EGEN is built with a compiler that supports 64 bit integers, and will allow the generation of C language expressions that include 64 bit integers. The type name of the 64 bit integer supported by the compiler used to build EGEN64 need not be the same as the name used by a compiler under test.

There are two predefined makefiles for building EGEN64 one under the dst-win directory for use in a 32 bit Windows™ environment and configured for the Microsoft 4.0 compiler, the other under the dst-ix directory for use in a UNIX™ environment configured for gcc. To build in either of these environments requires some configuration, as the default configuration is to build a standard EGEN without the 64 bit integer support.

Configuration of EGEN for 64 Bit Integer Support

If you wish to build EGEN with 64 bit support then the flags.h file in your destination directory needs this definition:

```
#define PH_INT64 1
```

Testing 64 BIT Integer Expression

Once EGEN has been built successfully if it is run with no option you should get the following output:

```
syntax: egen Version 2016a <switches> <opcodes>
   Compiled Jan 6 2016
   Configured with 64 bit integer support
   -D<database> (required) name of data base file
   -U           (-U or -V required) use unsign preserving typing rules
   -V           (-V or -U required) use value preserving typing rules
   -A           (-A or -K required) use ANSI typing rules for shift
   -K           (-K or -A required) use K&R typing rules for shift
   -R<number>   (optional) random number seed
   -S<name>     (optional) subroutine name
   -N<number>   (optional) number of statements
   -O<name>     (optional) output file name
   -I<name>     (optional) input file name
   -X           (optional) emit checking code at end
   -C           (optional) emit checking code after each statement
   -P           (optional) generate portable expressions
```

Note specifically the line indicating that 64 bit integer support has been enabled.

Using EGEN 64

This version of EGEN will now accept the keyword int64_t . For example, given the file int64.gen which contains the following:

```
auto   unsigned int64_t uxl = 70000;
static unsigned int64_t * puxl = &uxl;
auto   unsigned int64_t ** ppuxl = &puxl;
```

```
   auto   int64_t xl = 97;
   static int64_t * pxl = &xl;
   auto   int64_t ** ppxl = &pxl;

   register  int64_t rint1 = 1;
   register  int64_t rint2 = 2;
   register  int64_t rint3 = 3;
   register  int64_t rint4 = 4;
   register  int64_t rint5 = 5;
```

The operation

```
   egen $(EGENFLAG) -R23 -D$(SD)int64.gen -N10 "(+)" "*" "(-)" >test4.c
```

will generate a file as follows:

```
/*****************************************************************
 **    Self-checking C source code generated by EGEN component of  **
 **    The Plum Hall Validation Suite for C.                       **
 **    (C) 1986-1997 Plum Hall Inc                                 **
 **    EGEN Version 8.00                                           **
 **    This version supports 64 bit integers configured by         **
 **            inttypes.h                                          **
 **    EGEN -A -V -P -R23 -Df:\suite\testing\egen\int64.gen        **
 **            -N10 (+) * (-)                                      **
 *****************************************************************/

#include "defs.h"
#include "int64.h"

int main()
    {
        extern char *Filename;
        auto unsigned INT64 uxl;
        static unsigned INT64 * puxl;
        auto unsigned INT64 ** ppuxl;
        auto INT64 xl;
        static INT64 * pxl;
        auto INT64 ** ppxl;
        register INT64 rint1;
        register INT64 rint2;
        register INT64 rint3;
        register INT64 rint4;
        register INT64 rint5;

        /* ... rest of source file ... */
```

The generated files will contain the macro `INT64` as the name of your 64 bit integer type which you will need to define in your `int64.h` or `inttypes.h` header file.

STRESS

The STRESS section is a collection of EXIN scripts and data sets for EGEN.

The stress.ex script is intended to be run in the background on a multi-tasking operating system (or during programmer sleep-time, on a single-user system).

Each time it is started, it will execute repetitively; the number of iterations is specified by the variable $ITERATIONS. Periodically the output can be checked to see if any compiler errors have been detected. The allops.ex script will cycle through all of the C operators in conjunction with a second operator named on the command line. These two scripts are intended as examples of the kinds of tests that can be run using EGEN.

There are 3 data sets provided with the STRESS section:

integer.gen	all integer data
real.gen	all floating point data
mixed.gen	a mixture of real and floating data

New scripts can be adapted as necessary. If, for example, a compiler is having trouble with embedded assignment statements, a script can be run in the background with a statement like

```
egen -Dinteger.gen -N10 -R$STRESS @ "(" "{" += -= /= %= "}" ")" @
```

where the $STRESS variable is changed every time in a loop.

The stress.ex script can be modified for a different number of iterations; change the initializer of $ITERATIONS (around line 38).

COVER

Once the EXIN interpreter is built, it can be used to run the scripts in the COVER section. These scripts generate exhaustive coverage of simple expressions in the C language.

At the core of the COVER section is an EXIN script which, given two data sets and a C language operator, generates all possible permutations. All C operators (unary, binary, and ternary) can be covered with this script.

Command Line

The command line used to run this script is

```
exin <script> [-RESTART[<def>... ]] [<def>... ] [-A] [-X <extension1>
<extension2>]
```

where <def> is

```
<operator> | <data_set> [-S]
```

The script argument defines the name of the script file.

The -RESTART option (and its required arguments) allow these scripts to be restarted from within their pattern of test files.

The def arguments represent required combinations of operator and data set as defined by the script file. In general, you will never have more than two data sets and one operator in any one command line (including the arguments to the -RESTART option.)

The –X command line argument is described after the explanation of how cover works.

data_set represent required data set arguments, and is the name of one of the data sets from the table below. The optional argument -S tells the script to declare the variables from that data set as static rather than auto (the default).

operator represents an operator name. This is one of the C language operators from the table in the Cover Operators section below.

The final argument, -A, is also optional. If present, the output of the test is appended to a log file. The default is to create a new file.

Data Sets and Operators

There are two key terms fundamental to understanding the operation of the `COVER` scripts; "Data Sets" and "Operators".

A Data Set is a collection of data declarations and initializations used in the generation of a self-checking C program. The scalar data set, for example, contains declarations for:

```
char, unsigned char, signed char
short, unsigned short
int, unsigned int
long, unsigned long
float, double, long double
```

Other sets can be added as needed, but the current list of Data Sets is:

`scalar`	scalar data types
`pscalar1`	pointers to scalar data types
`pscalar2`	pointers to pointers to scalar data types
`union`	unions of scalar types
`punion`	pointers to unions of scalar types
`struct`	structure members
`pstruct1`	pointers to structure members
`pstruct2`	pointers to structures with pointers to structures
`array1`	one dimensional arrays of scalar types
`array2`	two dimensional arrays of scalar types
`bits`	bitfields
`pbits`	pointers to bitfields
`func`	function returning scalar types
`funcrp`	function returning pointer to scalar type
`funcrs`	function returning structure of scalar types
`arrarr`	array of scalars indexed by array of `int`

Cover Operators

Operators are the C language operators. Each of these operators is known by its name, such as `plus`. The cover script operators are:

`plus`	binary `+`	`not`	unary `!`
`minus`	binary `-`	`compl`	unary `~`
`times`	binary `*`	`preinc`	unary `++X`
`div`	binary `/`	`predec`	unary `--X`
`rem`	binary `%`	`postinc`	unary `X++`
`Lt`	binary `<`	`postdec`	unary `X--`
`Gt`	binary `>`	`quest`	ternary `? :`
`Le`	binary `<=`	`pluseq`	binary `+=`
`Ge`	binary `>=`	`minuseq`	binary `-=`
`eq`	binary `==`	`timeseq`	binary `*=`
`ne`	binary `!=`	`diveq`	binary `/=`
`andif`	binary `&&`	`remeq`	binary `%=`
`orelse`	binary `||`	`bandeq`	binary `&=`
`band`	binary `&`	`oreq`	binary `|=`

or	binary |	xoreq	binary ^=
xor	binary ^	lsheq	binary <<=
lsh	binary <<	rsheq	binary >>=
rsh	binary >>	cast	unary (TYPE)
uminus	unary -	assign	binary =

Each program generated by the COVER scripts reports errors in this form:

```
auto scalar auto scalar plus at line 234: (12) != (13)
```

Each program also prints a summary of the form:

```
***** 999 successful tests in auto scalar auto scalar plus *****
***** 2 errors found in auto scalar auto scalar plus *****
***** 0 skipped sections in auto scalar auto scalar plus *****
```

COVER Scripts

The COVER section contains scripts which allow the generation of C programs which check all possible permutations of the following:

cover.ex	2 data sets with any operator
alldata.ex	all data sets for one operator
allops.ex	2 data sets with all operators
all.ex	all data sets for all operators
sample.ex	a sampling of all data sets and operators

The syntax of each is:

```
exin cover.ex [-X <extension1> <extension2>] <d1> [-S] <d2> [-S] <op> [-A]
exin allops.ex [-RESTART <op>] [-X <extension1> <extension2>] <d1> <d2>
exin alldata.ex [-RESTART <d1> <d2>] [-X <extension1> <extension2>] <op>
exin all.ex [-RESTART <d1> <d2> <op>][-X <extension1> <extension2>]
exin sample.ex [-RESTART <d1> <op>]
```

Here is an example of the kind of program generated by the COVER scripts. The data sets were chosen as scalar vs. scalar, and the operator is plus (binary +).

```
#include "types.h"
int main()
{
extern char *Filename;
auto CHAR Ac = 7;
#if ANSI
auto SCHAR Asc = 8;
#endif
auto SHORT As = 9;
auto INT Ai = 10;
auto UCHAR Auc = 11;
auto USHORT Aus = 12;
auto UINT Aui = 13;
auto LONG Al = 14;
auto ULONG Aul = 16;
auto FLOAT Af = 16;
auto DOUBLE Ad = 17;
#if ANSI
auto LDOUBLE Ald = 18;
#endif
 /* a second distinct data set would go here */
```

```
Filename = " auto scalar auto scalar plus ";
iequals(__LINE__, Ac + Ac, 14 );
iequals(__LINE__, Ac + Ac, 14 );
#if ANSI
iequals(__LINE__, Ac + Asc, 15 );
iequals(__LINE__, Asc + Ac, 15 );
#endif
iequals(__LINE__, Ac + As, 16 );
iequals(__LINE__, As + Ac, 16 );
iequals(__LINE__, Ac + Ai, 17 );
iequals(__LINE__, Ai + Ac, 17 );
iequals(__LINE__, Ac + Auc, 18 );
iequals(__LINE__, Auc + Ac, 18 );
iequals(__LINE__, Ac + Aus, 19 );
iequals(__LINE__, Aus + Ac, 19 );
iequals(__LINE__, Ac + Aui, 20 );
iequals(__LINE__, Aui + Ac, 20 );
lequals(__LINE__, Ac + Al, 21L);
lequals(__LINE__, Al + Ac, 21L);
lequals(__LINE__, Ac + Aul, 22L);
lequals(__LINE__, Aul + Ac, 22L);
dequals(__LINE__, Ac + Af, 23.);
dequals(__LINE__, Af + Ac, 23.);
}
```

Non-Standard keywords (near, far)

The "–X" command line argument allow insertion of non-standard declaration modifiers, in particular "near" and "far". The cover.ex script generates all possible combinations of an operator and the 2 named data sets. If –X is included the first data set will be modified according to the first extension, and so on for the second. For example,

exin cover.ex –X near far scalar pscalar

will generate statements of the form:

```
int near Ai = 1;
int far Bi = 2;
int far * pBi = & Bi;
…
iequals(__LINE__, *pBi + Ai, 3);
```

Either of the –X arguments can be set to "" (nothing). Alternately, the –X construct is optional and can be left off. The same syntax and rules apply to the higher level scripts allops.ex and all.ex.

Configuring COVER

The `cover.ex` script reads a set of definitions from a file named `config.ex` for configuring the `COVER` section. The `makefile` for `COVER` copies the `exin/config.ex` configuration file into the `COVER` directory. (This simplifies the location of script files.) The configuration of `config.ex` that was done in the `EXIN` directory should not need any changes in `COVER`.

The GENERATE Option

By setting the configuration variable `GENERATE` to be `YES`, the test files will be generated, but not compiled, linked, or executed. (This was discussed earlier in the `EXIN` chapter.)

A Note on Naming

The files generated by `COVER` have rather cryptic names. This was necessary in order to guarantee unique names for all files under the `GENERATE` option, and still create names that are legal filenames on systems with name length limitations. The first two letters of the name are an encoding of the operator, the next letter encodes the first data set, followed by `s` (static) or `a` (auto), followed by an encoding of the second data set, and another `s` or `a`.

For example, the command line

```
exin cover.ex scalar pscalar1 -S plus
```

generates a file named `plaabs.c` (i.e., "`plus`, #1 data set, `auto`, #2 data set, `static`").

The operator mapping is done in the script `shname.ex`, and the data name mapping is done in the script `dnumbers.ex`.

The utility program `recvx` (built in the `EXIN` directory) reverses the mapping, so executing

```
recovx plaabs
```

produces this output line:

```
exin cover.ex scalar pscalar1 -S plus
```

If "char" is unsigned

In the cover directory the file config.ex contains the default line:

```
set UCHAR = N
```

This generates test code that treats "char" as a signed variable type. However, if this line is, instead, set to

```
set UCHAR = Y
```

the generated code will treat "char" as an "unsigned char".

LIMITS

The purpose of the `LIMITS` section of the Suite is to determine the value of certain compile time limits.

The ANSI/ISO Standard specifies a set of "minimum maximums" that a conforming implementation must meet. (See Section 5.2.4.1 of the Standard.) This section contains a set of scripts that determine the actual value of these limits (beyond the minimum requirement).

Running LIMITS

The `EXIN` script `limits.ex` calculates the actual limits for each of the required parameters. The script is invoked as

```
exin limits.ex <limit_name>
```

where `limit_name` is one of these Standard-required environment limits:

`blknest`	control structure nesting (15)
`condnest`	conditional compilation nesting (6)
`declmod`	declarator modifiers (12)
`dparens`	levels of declaration parenthesis nesting (31)
`parens`	levels of parenthesis nesting (32)
`iident`	characters of significance, internal identifier (31)
`eident`	external identifier name length (6)
`exid`	number of external identifiers in a file (511)
`blockid`	number of identifiers in a single block (127)
`macros`	number of macros simultaneously defined (1024)
`fparms`	number of parameters to a function call (31)
`mparms`	number of parameters in a macro (31)
`line`	number of characters in a logical line (509)
`string`	number of characters in a string (509)
`object`	bytes in an object file (32767)
`incnest`	include file nesting depth (8)
`cases`	case labels in a switch (257)
`members`	members in one structure or union (127)
`enums`	enumeration constants in one enumeration (127)
`stnest`	levels of structure nesting (15)

[Note: The minimums are not updated for C99 yet.] The `all.ex` script probes all of the ANSI limits. It is invoked as

```
exin all.ex
```

(Again, `make all` will accomplish the same thing.)

Note: The syntax generated into `nparms.c` may require a full standard compiler to handle the lines spliced with \. If your compiler cannot handle arbitrary line-splicing, `all.ex` will report an absurdly low capacity for macro parameters.

An Example Session with the Testing Section

The CONFORM section of the Plum Hall Validation Suite tests the compiler for conformance to the ANSI and ISO C standards. Since conformance to the standard is not a complete judgment of correctness, the TESTING section contains a few tools to help test for compiler quality.

At the heart of these tests are two programs: EXIN (EXecutive INterpreter) is a script language or "shell" similar the UNIX shell, but provided in portable C source code. EGEN (Expression GENerator) is a tool for generating C expressions and their expected results.

The directories of the TESTING section are:

EXIN	the source code of the script language interpreter.
EGEN	the source code of the expression generator
EGEN64	a destination only directory which allows building of a version of egen that suppor bit integers configured via the header file inttypes.h
LIMITS	a set of EXIN scripts which test the compiler for adherence to ANSI/ISO standard required performance limits.
COVER	a set of EXIN scripts that generate, build, and execute an exhaustive set of tests of operators and data types.
STRESS	an EXIN script that works with EGEN to test random expressions of arbitrary complexity.

Getting Started

The first step in using the TESTING section is to adapt two configuration files to the host system. In the root of the destination tree (e.g., dst-ix on a UNIX system, or dst-win on an WIN32 system) is the file flags.h. This header file is used when building EXIN to determine some of its characteristics.

Note: EXIN can be built with a completely empty flags.h and all of the scripts will work correctly. However, some of the options allow EXIN to use extra features for more convenient testing.

Under ANSI/ISO C, the system function is used to execute external programs. The standard does not require the exit code from the child program to be passed back to the caller. If you use the default (empty) flags.h, then child program failures can only be detected if the system function indicates them. On most systems there are other (non-ANSI/ISO) ways of invoking child programs that give good status indications. If your compiler is a UNIX or UNIX variant system, then be sure that flags.h contains

```
#define fooNIX 1
```

If your system is an MSDOS based system (including WIN85 and WINNT), then use

```
#define MSDOS 1
```

These flags enable the reliable result reporting.

The second file to be configured is config.ex. This is used by the COVER and STRESS tests. The variables to be configured are:

`set GENERATE = NO`
YES means generate test files, but do NOT compile, link, or execute

`set IF_ANSI = YES`
YES indicates that the compiler is ANSI/ISO Standard C.

`set DO_CLEANUP = YES`
YES says to delete the generated test files after executing them. *Note: COVER can use over 300 Megabytes with this set to NO.*

`set UCHAR = N`
indicates whether the char type is unsigned. N means no, Y means yes.

Running the Testing Section

The simplest way to run the testing section is to execute the `maketest` shell script (on UNIX systems) or the `maketest.bat` batch file on MSDOS. It will navigate all of the `TESTING` directories, "make" the programs, and run the tests. If all of this runs successfully, look at the file `testing.sum` for the results.

There are sections of this user guide that explain each of the `TESTING` sections in depth. This section will take a little tour of each directory and look at the generated output.

EXIN

After the EXIN interpreter is built, it is tested with a script that exercises its major features. The file `testall.out` shows the results of these tests. Each couple of lines shows the output of the test and a remark about what should be seen.

For example:

```
****** nested FOR/IF with GOTO ******
2 and 2
3 and 2
3 and 4
```

should see

```
2 and 2
3 and 2
3 and 4
```

The rest of the file contains similar tests and expected results.

The file `comptest.ex` exercises the commands in `config.ex` for compiling, linking, and executing a test program. In its output `comptest.out` you should see

```
COMPTEST passed
***** 1 successful test in COMPTEST *****
***** 0 errors detected in COMPTEST *****
***** 0 skipped sections in COMPTEST *****
```

This indicates that the script executes correctly.

EGEN

If EGEN was built successfully, then the files `test1.out, test2.out,` and `test3.out` will show this:

```
--- compile test1 ---
--- link test1 ---
***** Reached first test *****
***** 10 successful test cases in .\test1.c *****
***** 0 errors detected in .\test1.c *****
***** 0 skipped sections in .\test1.c *****
```

This shows that `EGEN` successfully generated, compiled, linked, and executed the test program. (Note: if your implementation produces 3-digit exponents like E+000, then the build test will show a few trivial diffs like “E+000” versus “E+00”. If that’s the only difference, then your EGEN is built ok.)

LIMITS

The limits section tests whether the compiler handles all of the program size limits of the ANSI/ISO standard. The file `all.out` indicates the results of this test.

Each of the tests has an indication of success or failure:

```
blknest :
**** MAXIMUM blknest is >= 128 (PASSED) ****
***** 1 successful test in blknest *****
```

or

```
blknest :
echo "**** MAXIMUM blknest is 12 (FAILED) ****"
echo "***** 1 error detected in blknest *****"
```

COVER

The COVER section tries to cover all operands and a rich set of data types by generating all of the possible permutations. The results of the first test are stored in `tryout.out`:

```
cover: asoapa auto funcrs auto arrarr assign
command /c compiler asoapa ./
cover: asosps static funcrs static arrarr assign
command /c compiler asosps ./
***** Reached first test *****
***** 288 successful test cases in asoapa.c ( auto funcrs auto arrarr assign ) ***
***** 0 errors detected in asoapa.c ( auto funcrs auto arrarr assign ) *****
***** 0 skipped sections in asoapa.c ( auto funcrs auto arrarr assign ) *****
***** Reached first test *****
***** 288 successful test cases in asosps.c ( static funcrs static arrarr assign )
***** 0 errors detected in asosps.c ( static funcrs static arrarr assign ) *****
***** 0 skipped sections in asosps.c ( static funcrs static arrarr assign ) *****
```

Now the COVER section is ready to run the major coverage test, `all.ex`. See the COVER description later in this user guide for more information.

STRESS

The stress section uses the EGEN expression generator to generate random expressions of arbitrary complexity. The output of the test is in `logfile`:

```
***** Reached first test *****
***** 10 successful test cases in ./int1.c *****
***** 0 errors detected in ./int1.c *****
***** 0 skipped sections in ./int1.c *****
...
```

The STRESS section of the User Guide explains how to configure the `stress.ex` script to generate different patterns of expressions.

Appendix 1: “Simple Benchmarks for C Compilers”, May 1988.

[The following article appeared in "C Users Journal" May 1988.
It describes the purpose and use of the enclosed benchmarks.
Also see The20thAnniversary.pdf in the bench subdirectory.]

SIMPLE BENCHMARKS FOR C COMPILERS

by Thomas Plum

Dr.Plum is the author of several books on C, including Efficient C (co-authored with Jim Brodie). He is Vice-Chair of the ANSI X3J11 Committee, and Chairman of Plum Hall Inc, which offers introductory and advanced seminars on C.

We are placing into the public domain some simple benchmarks with several appealing properties:

They are short enough to type while browsing at trade shows.

They are protected against overly-aggressive compiler optimizations.

They reflect empirically-observed operator frequencies in C programs.

They give a C programmer information directly relevant to programming.

In Efficient C, Jim Brodie and I described how useful it can be for a programmer to have a general idea of how many microseconds it takes to execute the "average operator" on register int's, on auto short's, on auto long's, and on double data, as well as the time for an integer multiply, and the time to call-and-return from a function. These six numbers allow a programmer to make very good first-order estimates of the CPU time that a particular algorithm will take.

The following easily-typed benchmark programs determine these times directly. The first one is benchreg.c ("benchmark for register operators"):

```
 1  /* benchreg - benchmark for  register  integers
 2   * Thomas Plum, Plum Hall Inc, 609-927-3770
 3   * If machine traps overflow, use an  unsigned  type
 4   * Let  T  be the execution time in milliseconds
 5   * Then  average time per operator  =  T/major  usec
 6   * (Because the inner loop has exactly 1000 operations)
 7   */
 8  #define STOR_CL register
 9  #define TYPE int
10  #include <stdio.h>
11  main(ac, av)
12          int ac;
13          char *av[];
14          {
15          STOR_CL TYPE a, b, c;
16          long d, major, atol();
17          static TYPE m[10] = {0};
18
19          major = atol(av[1]);
20          printf("executing %ld iterations0, major);
21          a = b = (av[1][0] - '0');
22          for (d = 1; d <= major; ++d)
23                  {
24                  /* inner loop executes 1000 selected operations */
25                  for (c = 1; c <= 40; ++c)
26                          {
27                          a = a + b + c;
28                          b = a >> 1;
29                          a = b % 10;
30                          m[a] = a;
31                          b = m[a] - b - c;
32                          a = b == c;
33                          b = a | c;
34                          a = !b;
35                          b = a + c;
36                          a = b > c;
37                          }
38                  }
39          printf("a=%d0, a);
40          }
```

If you enter this and compile it to produce an executable program, you can invoke it with one argument, the number of iterations for the major loop:

```
benchreg  10000
```

If this execution takes 16 seconds, this means that the average register operation takes 1.6 microseconds (16,000 milliseconds divided by 10,000 iterations of the major loop).

Let us examine the program in detail. Lines 8 and 9 define STOR_CL ("storage class") and TYPE to be register and int . Thus, on line 15, three variables (a , b , and c) are declared to be of this storage class and type. At line 16, the major loop control variables are long integers, but they are touched only one one-thousandth as often as the inner loop

variables, so they have little effect upon the timings. We are declaring the atol function to return a long integer; it would otherwise default to an int return. (If we were using a compiler based upon draft ANSI C, we could #include <stdlib.h> to get the declaration of atol , but this would limit the applicability of the benchmarks. This simple declaration is all that even an ANSI compiler would need.)

At line 19, we set the major loop variable to the number given on the command line, and at line 20, we confirm it to the output.

Line 21 is crucial to preventing some overly aggressive optimizations. Earlier versions of these benchmarks had simply initialized a and b to 1, but this allows a compiler to forward-propagate a known constant value. The expression av[1][0] gives the first digit-character of the command-line argument; subtracting '0' produces a digit between 0 and 9. (Yes, the latest ANSI draft now guarantees that the digit characters are a contiguous sequence in any environment.)

Line 22 simply executes the major loop the number of times given by the variable major . Line 25 repeats the inner loop 40 times, and with 25 operators in that loop, this produces 1000 operators. (Actually there are 1003, because of the initialization and the extra increment and test at loop completion. The discrepancy is well within acceptable tolerances.)

Within the inner loop, 40% of the operators are assignments, in keeping with the percentages reported in the original Drhystone work. Of the other operators, the most frequent are plus and minus. The sequence of operations is carefully chosen to ensure that a very aggressive optimizer cannot find any useless code sections; each result depends functionally upon previous results.

Finally, the printout at line 39 is also important to preventing over-optimization. If the compiler could notice that we did nothing with the computed result, it could discard all the operations that produced that result.

We have completed our perusal of the first benchmark program, benchreg.c . The second program (benchsho.c , for short's) is derived from benchreg.c by changing lines 8 and 9: STOR_CL becomes auto , and TYPE becomes short . The program is otherwise unchanged.

The third program (benchlng.c , for long's) is obtained by leaving STOR_CL as auto and changing TYPE to long .

To make the fourth program (benchmul.c , for multiplies) we set TYPE to int , and change lines 27 through 36 to one source line which does 25 multiplies:

```
a = 3 *a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a; /* 25 * */
```

The fifth program (benchfn.c , for functions) is a major rewrite. We arrange a series of function definitions for f3 , f2 , f1 , and f0 such that each call to function f0 generates exactly 1000 function-call operations. In case the compiler has an aggressive optimizer, move the function f3 to a separate source file, so that the compiler cannot see how useless

it is. The global variable dummy will make the compiler think that f3 might be up to something useful. Here, then, is the benchfn.c function:

```
 1  /* benchfn - benchmark for function calls
 2   * Thomas Plum, Plum Hall Inc, 609-927-3770
 3   * Let  T  be the execution time in milliseconds
 4   * Then  average time per operator  =  T/major  usec
 5   * (Because the inner loop has exactly 1000 operations)
 6   */
 7  #include <stdio.h>
 8  int dummy = 0;
 9
10  /* f3 - lowest level function
11   * Put this in separate source file if compiler detects and
12   * optimizes useless code
13   */
14  f3() { }
15
16  f2() { f3();f3();f3();f3();f3();f3();f3();f3();f3();f3();} /* 10 */
17  f1() { f2();f2();f2();f2();f2();f2();f2();f2();f2();f2();} /* 10 */
18  f0() { f1();f1();f1();f1();f1();f1();f1();f1();f1();} /* 9 */
19
20  main(ac, av)
21          int ac;
22          char *av[];
23          {
24          long d, major, atol();
25
26          major = atol(av[1]);
27          printf("executing %ld iterations0, major);
28          for (d = 1; d <= major; ++d)
29                  f0(); /* executes 1000 calls */
30          printf("dummy=%d0, dummy);
31          }
```

The sixth program (benchdblc. , for double's) is derived from benchlng.c by changing STOR_CL to auto , TYPE to double , and replacing the inner loop body with this slightly different version:

```
a = a + b + c;
b = a * 2;
a = b / 10;
a = -a;
b = -a - b - c;
a = b == c;
b = a + c;
a = !b;
b = a + c;
a = b > c;
```

These changes are necessary because floating-point operands are not allowed for the shift, remainder, and bitwise operators, and because the subscript operator does not really exercise the floating-point instructions. This revised inner loop still gives us a representative mix of typical operations.

This, then, completes our collection of six benchmark programs. After they are compiled to produce executable programs, the next question is "How do I time the execution?"

On UNIX systems, the timing is easy -- just run the time command:

```
$ time benchreg 10000
```

The sum of the "user" and "system" times will give the CPU time used by the program.

More accurately, we could time the execution of zero iterations, and subtract that time from the time for the measured number of iterations.

On MS-DOS systems, timings can be obtained, but with greater difficulty. If we create a file named CR-LF which contains just one newline (or "carriage-return-newline" in DOS parlance), we could time our program with a "batch" file such as this:

```
time <cr-lf
benchreg 0
time <cr-lf
benchreg 10000
time <cr-lf
```

We must then take times that are expressed in minutes-and-seconds and produce differences expressed in seconds.

With whichever method, we eventually produce six numbers that are characteristic of a particular environment (a specific compiler supporting a specific machine).

[NOTE: Since this article appeared, I have added a driver program, bench2.c. In an ANSI environment with the clock function, it will run all the tests and report the results, eliminating the need for manual computations. 91/10/01: I have deleted the int-multiply benchmark. Compiler vendors have begun to use ``benchmark-recognizers''. And anyway, multiply isn't very vendor-dependent.]

Here are some examples of timing results that have been obtained on a variety of minicomputer and workstation environments:

Machine/compiler	register int	auto short	auto long	int multiply	func call	auto dbl
AT&T 3B2/05 (-O)	1.36	3.87	2.62	15.4	7.7	22.5
AT&T 3B2/05 (no -O)	1.78	4.66	2.75	16.2	9.3	22.5
AT&T 3B2/400 (-O)	1.09	1.36	1.10	16.2	10.0(?)	91.4
AT&T 3B2/400 (no -O)	1.14	2.61	2.36	17.3	11.3	91.1
Apollo DN330 (-O)	1.36	.78	1.36	10.17	3.57	
Apollo DN330 (no -O)	1.54	1.28	1.54	11.30	3.64	
Apollo DN580 (-O)	1.03	.59	1.03	7.67	2.72	
Apollo DN580 (no -O)	1.18	.97	1.18	8.48	2.77	
Apollo DN660 (-O)	5.88	1.24	5.88	21.86	4.26	
Apollo DN660 (no -O)	5.93	1.52	5.93	21.93	4.29	
Cray X-MP (no vectors)	.0567	.0656	.0822	.366	.821	.082
Masscomp 5500	3.18	2.7	4.9	30.8	7.3	
Masscomp 5600 (-O)	.45	.61	.46	2.83	1.04	
Masscomp 5600 (no -O)	.46	.78	.64	2.99	1.76	
Pyramid 90X (-O)	.85	1.04	.86	3.64	1.9	2.37
Pyramid 90X (no -O)	.86	1.01	.86	3.65	1.8	2.34
Sequent (-O)	1.39	2.99	2.53	9.90	9.3	
Sequent (no -O)	1.50	3.25	2.83	9.95	13.2	
Sun 3/260HM (-O)	.31	.48	.47	1.98	1.16	
Sun 3/260HM (no -O)	.36	.58	.57	1.99	1.62	
Sun 3/75M (-O)	.47	.77	.76	3.00	2.12	
Sun 3/75M (no -O)	.53	.95	.94	3.01	2.73	
Sun 3/75M(4.2, -O)	.50	.81	.83	2.85	1.5	20.7
Sun 3/75M(4.2, no -O)	.54	1.00	1.01	2.97	2.7	21.1
Sun 3/75M(VM, -O)	.46	.77	.75	2.96	2.1	20.8
Sun 3/75M(VM, no -O)	.52	.96	.93	2.97	2.7	21.1
VAX 11/730 (-O)	4.00	9.80	6.20	16.2	42.8	12.4
VAX 11/730 (no -O)	4.73	10.2	7.45	16.57	51.5	17.0
VAX 11/780 (-O)	1.21	2.43	1.67	2.76	15.0	2.95
VAX 11/780 (BSD 4.2)	1.38	2.42	1.96	2.92	17.2	
VAX 11/780 (UNIX 5.2)	1.24	2.48	1.79	2.72	15.7	3.89
VAX 11/780 (no -O)	1.29	2.51	1.85	2.70	16.7	3.89
VAX 11/785 (-O)	.93	1.85	1.32	5.00	13.9	47.5
VAX 11/785 (no -O)	1.01	1.96	1.44	5.08	14.2	5.42
VAX 8650(UNIX -O)	.236	.484	.298	.589	2.63	.578
VAX 8650(UNIX no -O)	.258	.482	.316	.574	3.06	.791
VAX 8650(Ultrix -O)	.23	.40	.29	.53	2.4	.56
VAX 8650(Ultrix no -O)	.26	.41	.34	.56	2.8	.77

Notice that some of these timings were run before the benchdbl benchmark had been written. There are no examples of the popular PC environments in this table. If interested readers wish to run these benchmarks on their own environments, I will endeavor to present these results in a future article.

Processor speeds are sometimes described in "MIPS" (millions of instructions per second); using a value such as the number of register operators per second in C might give rise to a "MOPS" measurement of more use to C programmers. Those of us who have tried these benchmarks have appreciated the intuitive grasp that they give of the speed of current machines and compilers. I hope that you too will find them of interest.