Testing C++ Compilers for ISO Language Conformance

Conforming to standards really is a big deal

Brian A. Malloy, Scott A. Linde, Edward B. Duffy, and James F. Power

Conformance to a standard is one of the most important assurances compiler vendors can make. Conformance can affect acceptance of the compiler and, in many cases, impact the language itself. Conformance enables code portability and wider use of the language and corresponding libraries. And even if code portability isn't important, conformance facilitates documentation so that text books and language-reference manuals have a common frame of reference.

However, conformance is especially important and difficult for C++ because the language standard was slow to develop and acceptance of the standard occurred years after its introduction. By 1998, when the ISO Standard was accepted, there were many established and accepted C++ compilers in use.

In this article, we describe a test harness we built to measure conformance of C++ compilers. In applying the same standard to all vendors under consideration, we use the same test cases and testing framework for all executions—even though some of the compilers are platform dependent and there is no common platform for all compilers. We found that with its scripting facility, platform independence, and object orientation (facilitating code reuse), Python provided the functionality we needed. Moreover, unlike other languages, Python includes a testing framework with the language. This testing framework is a Python module called "*unittest*" (http://pyunit.sourceforge.net/), written by S. Purcell, and patterned after the JUnit framework developed by Kent Beck and Erich Gamma (http://members.pingnet.ch/gamma/junit.htm), and included with Python 2.1 and later (http://www.python.org/). We have extended the framework to facilitate measurement of ISO conformance.

We use our extended framework in all test case executions. To avoid bias for or against any vendor, our test case selection is based on test cases found directly in the ISO C++ Standard (*International Standard: Programming Languages—C++*. Number 14882:1 998(E) in ASC X3. American National Standards Institute, First Edition, September 1998. ISO/IEC JTC 1). All of our test cases are actual examples listed specifically in the Standard with outcomes specified.

The Python code, together with Clause 3 test cases from the ISO Standard, are available electronically from *DDJ* (see "Resource Center," page 5) and http://www.cs.clemson.edu/~malloy/projects/ddj.html.

Brian Malloy is an associate professor in the computer science department at Clemson University and can be contacted at http://www.brianmalloy.com/. Scott Linde received his M.S. in Computer Science from Clemson University in February, 2002. This paper is part of his M.S. thesis. Edward Duffy is a masters student in the computer science department at Clemson University. James Power is a researcher in the computer science department at the National University of Ireland at Maynooth.

The *unittest* Testing Framework

In Python, each class begins with the keyword *class* and each function begins with the keyword *def*. There are classes defined on lines 1–3 of Listing One and functions defined on lines 4, 10, 12, 14, 16, 18, and 26. The classes in lines 1–2 do not contain data or methods, as indicated by the keyword *pass*: They are used to handle exceptions. Class *Binary* (lines 3–25) represents an abstraction for binary numbers. Users of the class instantiate *Binary* numbers using positive decimal numbers, then, using the overloaded operators, manipulate each binary number in the same manner as an integer, applying addition, multiplication, comparison, and output. *Binary* has six member functions. Function *__init__* (lines 4–9) is the constructor for *Binary*. Functions *__add__*, *__mul__*, *__eq__*, and *__ne__* overload the +, *, ==, and ! = operators for two binary numbers. Function *__str__* (lines 18–25) enables output of binary numbers, similar to operator << for C++, and *toString* for Java. Function *test()* (lines 26–34) represents one approach to testing class *Binary*. Binary numbers *bin*1 and *bin*2 are instantiated in lines 27–28, then used in addition, multiplication, and comparison in lines 29–34. Function *test()* is invoked (line 36) when the module is executed in standalone fashion.

These tests aren't intended to be exhaustive, but rather illustrate one approach to testing modules. There are drawbacks, of course, in using this approach. First, the code to test the module is included in the module itself, which can distract readers interested in understanding the class and requires that the code to test the module be shipped with the module. Second, the person performing the test must inspect the output and verify it is correct. If this verification were performed automatically with a summary report at the end of the test, the testing process would be less prone to error.

Listing Two is an alternative, using the *unittest* module, to the testing approach in Listing One. (For detailed information on *unittest*, see the PyUnit homepage, the *Python Library Reference*, and Mark

Understand Your Software...

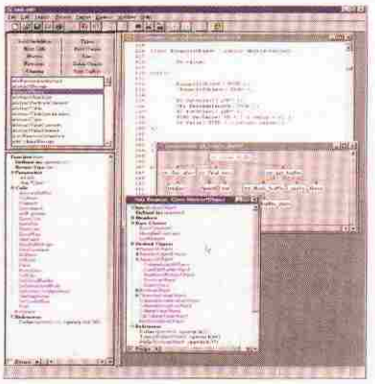
But I didn't write this... lump of spaghetti the boss just dumped on the server, or the source for a product my company picked up in a merger. Wherever it came from, it's big, it's ugly, it uses macros to make C look like Pascal, and I need help.

Help is available... *Understand for C++* helps C/C++ and Java developers understand, maintain, measure, and document code they didn't write, they wrote a long time ago, or that has grown too large or complex to comprehend.

How are we different?...Each tool quickly parses and manipulates very large amounts of code. 1,000,000 SLOC projects are common among our customers. We also focus on exceptional customer support with rapid response from a real engineer. Fixes and new features are incorporated into weekly builds.

See for yourself... **http://www.scitools.com/**

Understand for C++/Java



Now supports Java

- K&R C, ANSI C, C++, Java
- Analyzes all 4 languages together
- Fast on big projects
- "Explorers" programmer editor—nifty for exploring code
- Detailed cross reference of everything - one click visiting
- Detailed information for everything declared or used in code
- Works on partial code or missing preprocessor definitions
- Class browsing (inheritance and base)
- Graphical Call Tree, Call-By Trees, Include, Include-By Trees
- Visio export of graphics
- PERL or C API to get data/graphics for custom documentation
- Comment extraction for documentation generation
- Freely sharable HTML and text reports of analysis information
- Complexity, Size, OO, and other metrics,
- Easy export of metrics to spreadsheets and databases

Download and try for free at http://www.scitools.com

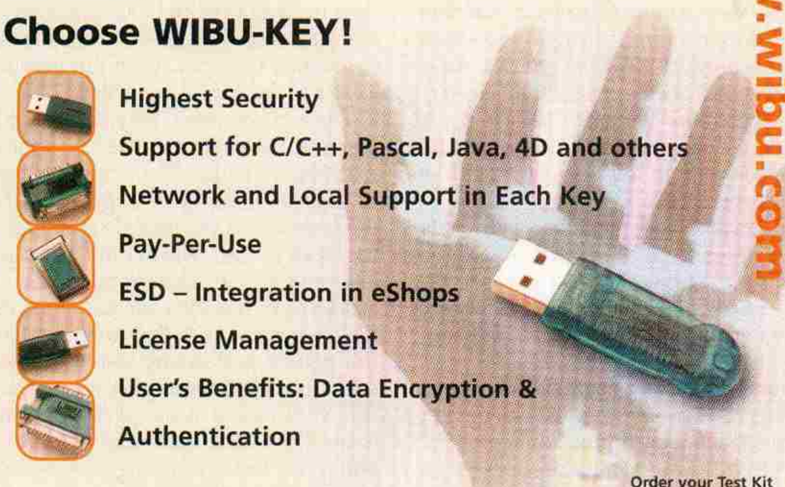
© Scientific Toolworks, Inc. 1-802-763-2995 info@scitools.com

SOFTWARE & DOCUMENT PROTECTION

It's all in your hands!

You need a reliable copy protection system that can be flexibly adapted to the requirements of your software sales strategy?

Choose WIBU-KEY!



Highest Security

Support for C/C++, Pascal, Java, 4D and others

Network and Local Support in Each Key

Pay-Per-Use

ESD – Integration in eShops

License Management

User's Benefits: Data Encryption & Authentication

www.wibu.com

Order your Test Kit 1-800-986-6578

WIBU SYSTEMS

WIBU-SYSTEMS AG
D-76137 Karlsruhe
WIBU-SYSTEMS USA, Inc.
Seattle, WA 98101
Email: info@wibu.com
Web: www.wibu.com

USA, Canada:
Griffin Technologies, LLC
Email: sales@griftech.com
Web: www.griftech.com

GRIFFIN TECHNOLOGIES

Argentina, Australia, Belgium, Canada, Croatia, Denmark, Egypt, Finland, France, Germany, Great Britain, Hungary, Japan, Jordan, Korea, Lebanon, Netherlands, Spain, Portugal, Syria, USA.

Pilgrim's public-domain book, *Dive Into Python* at http://diveintopython.org/.)

Listing Two imports the modules *unittest*, *Binary*, and *InvalidBinaryError* in lines 1, 2, and 3, respectively. The *import* format on line 1 requires all uses of the imported module be prefixed with the module name; the format used in lines 2–3 do not require the module name prefix. The classes *BinaryTest* (line 4) and *BadInputTest* (line 18) encapsulate the test cases we use to test class *Binary*. Both classes are derived from *TestCase*, a class in module *unittest*; this inheritance is indicated by putting the class names in parentheses at the point of declaration (lines 4–18). By inheriting from *TestClass*, we acquire useful methods to facilitate testing.

BinaryTest and *BadInputTest* encapsulate the testing process of *Binary*, with *BinaryTest* testing for success, and *BadInputTest* testing for failure. *BinaryTest* contains five methods that either initialize the test process, *test*, or recover from the test process. The first method in *BinaryTest*, *setUp* (lines 5–6), instantiates a *Binary* zero as a data member. Method *tearDown* (line 7) does nothing, but might be used to clean up after the test process. There are three test cases in *BinaryTest*: methods *testZero*, *testAddition*, and *testMultiplication*, each containing *assertEqual* statements to determine if the values returned by the *Binary* API are correct. The statement in line 9 of *testZero* compares the value of the data member *self.n* to the newly instantiated number *Binary(0)*: If they are equal, the test passes; if not, *assertEqual* raises an exception and the test fails. *testAddition* is one test case but actually tests two addition operations. The *testMultiplication* method is also a single test case, but tests 100 multiplication operations.

BadInputTest uses the *assertRaises* statement to ensure the *Binary* API handles bad input. The *testNegative* test case (line 21, Listing Two) ensures that *Binary* raises the exception *InvalidBinaryError* (line 23) if users of the API attempt to instantiate a negative binary number; this test case passes. However, *testDecimal*, the method to test for decimal input (line 25), does not pass since the *Binary* API does not raise an exception when users attempt to instantiate a decimal *Binary*. This is the only test case in Listing Two that fails.

unittest automatically calls *setUp* and *tearDown* before/after each test case execution, and the three test methods in *BinaryTest* are invoked automatically by *unittest*. Similarly, the *setUp* routine for *BadInputTest* and both its negative test cases are automatically called. In fact, a lot automatically happens when using Listing Two in standalone fashion. For instance, when *main* is invoked in line

29, all methods that begin with "test" in classes *BinaryTest* and *BadInputTest* are recognized as test cases and a test suite is constructed consisting of each of these methods. These test cases then run automatically; the order of execution is determined by a function that sorts the test cases lexicographically by the name of the function using the built-in Python *cmp* function. The testing framework provides an environment in which the test cases can execute and a report is generated. Alternatively, users can construct the test suite and pass the name of the test method as a parameter to the newly constructed test suite. Executing the tests in Listing Two produces the output in Figure 1, with five test cases executed and one test case failure (*testDecimal*).

Building the Test Harness

Listing Three, the test case generation module called "*runtests.py*," consists of two functions, *doTests* (lines 3–13) and *cleanUp* (lines 14–21). Each Python module contains a global namespace with an identifier called *name* that stores the module's name. When a Python interpreter session begins executing a module, the value of *name* is *main*. Thus, we begin our test case generation by running the Python interpreter on runtests.py and the *if* statement (line 22) evaluates to True.

When the session begins, the *if* statement (lines 23–34) verifies user input and calls functions to run the tests and clean up. Our test suite is partitioned into directories and we have a directory for each clause in the standard that we test. The *if* statement in line 23 verifies that two parameters were entered, and line 28 verifies that the second parameter is a valid directory within the current directory. Then, *doTests* and *cleanUp* do the testing of the clause and clean up afterward.

Function *doTests* (line 3) accepts two arguments: the full path to the directory containing the clause under test, *full-path*, and the directory name, *directory*. The function gathers a list of the files contained in the directory, and instantiates a test runner (using *TextTestRunner*) and a test suite (using *TestSuite*). *TextTestRunner* and *TestSuite* are part of the *unittest* framework, which we import. The *for* loop in *doTests* examines each name in the list to determine if it's a file, and verifies the proper extension. If the name represents a file with a C++ extension, a test case is generated (line 10) with two parameters passed to the constructor: the function that executes the test, *testExecute*, and the prefix of the name of the C++ test case. After the test case is generated, it is added to the test suite. The final action of *doTests* tells the *TextTestRunner* object, *runner*, to run the tests (line 13).

Function *cleanUp* (Listing Three, lines 14–21) cleans up after the test suite is executed. We could have used the *tearDown* method in *CppTestCase*, derived from *unittest* to clean up after individual test cases, but we found it more efficient to clean up after all test cases when the test suite has been executed. Line 15 gets a listing of the files in the directory of the clause under test and the *for* loop examines each file to see if it should be removed. In running each test case, we may have constructed an object file or executable and these are also removed as part of the cleanup process.

The C++ Test Case Wrapper

Listings Four and Five present class *CppTestCase*, a wrapper for our C++ test cases extracted from the ISO C++ Standard. The class contains four methods. Method *init* (lines 3–18) is the class constructor, and *setUp* (lines 19–28) performs initialization for each test case. Method *tearDown* does nothing because we recover from test case execution in the test case generator after the entire suite is executed, as previously described. Function *testExecute* is the longest method in the class and we show only its signature in Listing Four; the code for *testExecute* is in Listing Five.

The constructor for *CppTestCase* (*init* in Listing Four) initializes the data used in the test case. The method begins by calling the constructor of the superclass, *TestCase* (line 4), passing the name of the function that executes the test case; in our

On Time RTOS-32

On Time's royalty-free embedded RTOS for 32-bit x86 implements a Windows NT subset kernel in only 16k.

On Time RTOS-32 has a scalable component architecture. Only the components actually needed by the application are loaded on the target.

Application
RTKernel-32
RTFiles-32
RTIP-32
RTPEG-32
RTTarget-32
Target Hardware

On Time RTOS-32 Features:

- Source and binary compatibility with Windows 95/98/NT/2000/XP
- Supports DLLs
- Unmatched real-time performance
- Context switch 0.73µs on P120
- Supports (but does not require) PC compatible hardware
- Boots from disk, BIOS extension, ROM, or DOS
- Windows 95 look-and-feel GUI
- Free tech support
- Full source code available
- Supports Microsoft, Borland, Delphi Compilers
- **Fully integrates with MS Visual Studio 6/.NET**
- **Free Evaluation Kit** available at http://www.on-time.com
- **No run-time royalties**

On Time RTOS-32 Components:

RTTarget-32
Development System
Boot Code, Locator, Cross Debugging, Win32 Emulation Library
Price: $1500

RTKernel-32
Real-Time Multitasking Scheduler
Hard real-time, unlimited number of tasks, 64 priorities, 1µs time resolution
Price: $1500

RTFiles-32
Embedded FAT File System
FAT-12, FAT-16, FAT-32
For floppy disks, IDE hard and flash disks, M-Systems DiskOnChip, linear flash
Price: $1500

RTIP-32
Embedded TCP/IP Stack
BOOTP, RARP, ARP, ICMP, UDP, TCP, DNS
Add-Ons: PPP, FTP, TFTP, Web Server, SMTP/POP3, TELNET, SNMP, DHCP
Price: $3000

RTPEG-32
Embedded GUI
Windows 95 look-and-feel Graphical User Interface
Price: $2500

North America: On Time • 15 Spur Lane • Centereach, NY 11720 • USA
Phone (888) 667-8200 • Fax (631) 471-8850 • email info@on-time.com

International: On Time • Hofweg 49 • 22085 Hamburg • Germany
Phone +49-40-2279405 • Fax +49-40-2279263 • email info@on-time.de

http://www.on-time.com

On Time
REAL-TIME AND SYSTEM SOFTWARE

framework this is *testExecute*, introduced in Listing Three. It is important that the *CppTestCase* constructor explicitly calls the constructor of the superclass because, in Python, constructors for superclasses are not automatically invoked. Lines 5–14 initialize a list that contains the calls for each of the compilers we use to execute C++ test cases; the actual compiler is chosen in *testExecute* by indexing into this list. Included with the compiler call is a flag, passed using the *–D* option, that may be used in the test cases to determine the name of the files to include. We also set the name of the file for this C++ test case, the *toPass* flag that indicates if this test case is supposed to pass or fail, and *hasMain*, a flag that indicates if this test case has a main function. Finally, the directory is initialized to the current working directory, line 18.

Method *setUp* (lines 19–28, Listing Four) parses the test case to determine if it contains a function *main*. If it does, the *hasMain* flag is set to True (line 20) and the test case is compiled, linked, and executed. Listing Four is *testExecute*, the code for the final method of *CppTestCase*. The first part of *testExecute* chooses the compiler, link, and execute call, and then compiles the program. If the test case has a *main* and it successfully compiled, the program is linked and executed.

The system calls to compile, link, and execute the program are on lines 4, 7, and 10 of Listing Four, respectively, where the results are assigned to variables *compiled*, *linked*, and *executed*. All of the systems we used follow the convention that upon successful compile, link, or execute, a zero value is returned; otherwise, a nonzero value is returned. However, Windows 95/98 do not follow this convention, but return a zero value for both success and failure. Thus, our framework will not provide correct results on these systems.

Some examples in the ISO Standard are intended to compile, others to link, execute, and give a specified output, and still others are intended to fail. We translated the examples into test cases that are either positive or negative, depending on whether they are expected to successfully compile, link, or execute. Negative test cases are intended to expose compilers that accept a superset of the Standard. Thus, a negative test case does not pass if it compiles successfully, or compiles and executes successfully. Forty-one percent of the test cases we extracted from the Standard are negative test cases and form an important part of our measurement of ISO conformance.

We applied our testing framework to several C++ compilers running on several different platforms

The most important function of *testExecute* is to determine whether the test case passes or fails, based on the values of the flags *toPass* and *hasMain* and the outcome of the compile, link/execute phase of the test process. If the *toPass* flag is True, then this is a positive test case; if the *hasMain* flag is True, then this test case is supposed to link, execute, and possibly give a specified output. We make a judgment about whether the test case passes based on the values in these two flags and the outcome of the compile and link/execute phases of the test. Thus, there are four variables that must be modeled, producing 16 possible paths, six of which are infeasible. We model the 10 possible outcomes on lines 12–48 of Listing Five.

Test Case Extraction

A single example in the Standard can produce many test cases. Some examples expand into multiple positive test cases, while others may expand into a single positive test case and multiple negative test cases. Consider Listing Six, taken from Clause 3 of the ISO Standard, which specifies rules for name lookup in namespaces. Listing Six represents a single example in the Standard, but clearly this must be more than one test case. For example, there are errors on lines 12 and 17; if this example is used as a single test case and the program fails, the tester will not be able to determine if the error occurred on line 12, line 17, or both.

In our approach, we generate three test cases for Listing Six: One test case with no errors that should pass, another test case with the first error that should fail, and a third test case with the second error that should also fail. Thus, we get one positive and two negative test cases. However, a different testing approach might generate many more test cases than three using the example in Listing Six.

Many of the positive examples will not compile as described in the Standard. Some examples require variable or type declarations, or header file inclusion. We have found a wide variation in nomenclature of include files across vendors. In some cases, we were able to avoid the problem of this variation in the include file names if the class or function in the included file is not part of the test. For example, a variable declaration such as *strings*; might be modified to *ints*; if the purpose of the test does not involve the *string* class.

However, in some cases a function or class in the included file is part of the test. Listing Seven, taken from Clause 3 of the ISO Standard, illustrates a test case where the function *memcpy* is part of the test. In Listing Seven, *memcpy* is used to copy a value from a *struct* to a buffer, then back again to the *struct*; the test case succeeds if the value is successfully transferred in both directions. However, the Borland compiler places *memcpy* in *mem.h* while the other compilers place *memcpy* in *memory.h*. The conditionally compiled code, lines 1–5 of Listing Seven, chooses the file to include based on the compiler under test.

C++ Conformance Roundup

The article "C++ Conformance Roundup," by Herb Sutter (*C/C++ User's Journal*, April 2001), presents the results of a roundup of a dozen C++ compiler and library vendors in an attempt to establish their conformance to the ISO Standard. In the roundup, three suppliers of C++ conformance test suites—Dinkumware (http://www.dinkumware.com/), Perennial (http://www.peren.com/), and Plum Hall (http://www.plumhall.com/)—were asked to evaluate compilers from IBM, Sun Microsystems, Kuck and Associates, Metrowerks, Intel, Hewlett-Packard,

```
% python binarytest.py
F....
==============================================
FAIL: testDecimal (__main__.BadInputTest)
----------------------------------------------
Traceback (most recent call last):
File "binarytest.py", line 27, in testDecimal
self.assertRaises(InvalidBinaryError,Binary,0.5)
File "/usr/local/Python-2.2 /Lib/unittest.py",
   line 279, in failUnlessRaises
   raise self.failureException, excName
AssertionError: InvalidBinaryError
------------------------------------------------
Ran 5 tests in 0.019s
FAILED (failures=1)
```

Figure 1: *Executing the tests in Listing Two produces this output.*

(continued from page 74)
Microsoft, GNU, Borland (BC++ and BCC), and Comeau Computing.

The Plum Hall test suite is based on providing a test case for each sentence in the ISO Standard. For Clauses 1 through 16, describing the language definition, this line-by-line approach produced some 4356 test cases. Perennial has used a similar approach but produced nearly 10 times as many— a total of 35,993 test cases for the same clauses.

We found the Conformance Roundup to be inconclusive, so we decided to design our own conformance tests. In an attempt to factor out bias, we decided to use the same testing framework for all test executions. Moreover, rather than engage in a line-by-line interpretation of the ISO Standard, which might bias us toward the compiler with which we were most familiar, we have chosen to only extract explicit examples in the Standard with outcomes specified. Using this approach, we extracted 760 test cases.

Case Study

We applied our testing framework to several C++ compilers running on several different platforms. The compilers in our study include Borland 5.5.1 (http://www.borland.com/), Visual C++ 6.0 (http://www.microsoft.com/), gcc 2.95.2, gcc 2.96, and gcc 3.0.4 (http://gcc.gnu.org/), and MIPSpro7.3.1.2m (http://www.sgi.com/). We executed the test cases for Borland 5.5.1 and Visual C++ 6.0 on Windows NT/2000; the rest of the test cases were executed on Linux or Solaris systems running Red Hat 7.1 or Solaris SunOS 5.8. We have tested the framework on Python 1.5 through 2.2; for versions of Python prior to 2.1, the *unittest* module must be downloaded separately. To provide some insight into the efficiency of the Python framework, we were able to run the 217 test cases for Clause 14, containing the largest number of test cases, in 5.125 seconds on a Dell Precision 530 workstation, with a Xeon 1.7-GHz processor and 512 MB of Rambus memory.

Figure 2 summarizes our results, where the first column lists the names of the compilers and the columns labeled 3–15 list the results for Clauses 3–15 for the respective compilers. The column labeled "Failures" lists the total number of test cases failed by the respective compiler and the final column, "% Passed," represents the percentage of test cases that passed. The bottom row in Figure 2 lists the number of test cases in each of the respective clauses, with the total number of test cases at 760. For example, column 1 shows that the gcc 3.0.4 compiler failed 8 out of 88 test cases for Clause 3 of the ISO Standard.

The final column of Figure 2 shows that the first three compilers passed at least 90 percent of the test cases, with the gcc 2.95.2 and Borland compilers very close to 90 percent. Moreover, the Visual C++ 6.0 compiler also performed well in the tests. Our goal here is to show that our testing framework is extensible to multiple platforms and provide some measure of how the compilers stack up against the examples in the ISO Standard. We are not considering compile speed, efficiency of optimized code, or friendliness of the environments. Moreover, the performance of a given compiler on these tests may not directly predict the performance of the compiler on a real-world program or test suite. To underscore the intricacy of the test cases, consider Listing Eight, a test case that all compilers failed. The purpose of the test case is to illustrate that the expression in line 6 is not a function call and that argument-dependent name lookup does not apply; rather, the expression is a cast equivalent to int(a). However, the compilers we tested find the expression in line 6 to be a redeclaration of the *friend* function in line 3 and became confused with the *typedef* in line 1. None of the compilers were able to compile this example correctly.

Figure 2 provides an overview of compliance on a clause-by-clause basis. For example, the four bars at the top of the graph illustrate the percentage of Clause 15 test cases passed by gcc 3.0.4, MIPSpro7.3.1.2m, Borland 5.5.1, and Visual C++ 6.0. The bar in the graph for Clause 4 (Conversions) shows that gcc 3.0.4 and Borland 5.5.1 passed both tests, while

New Version

PC-lint 8.0 for C/C++

Bug of the Month #682

```
#include <stdio.h>

int an[4] = { 1, 2, 3, 4 };

int sum( int a[4] )
    {
    int limit = sizeof(a) / sizeof(int);
    int result = 0;
    for( int i = 0; i < limit; i++ )
        result += a[i];
    return result;
    }

int main()
    {
    printf( "%d\n", sum( an ) );
    return 0;
    }
```

The programmer expected that summing the elements of the array would produce the value 10. Instead he got something much smaller. What did he do wrong? Call if you need a hint or visit our web site at **www.gimpel.com**

PC-lint for C/C++ will catch this and many other bugs. It will analyze a mixed suite of C and C++ modules to uncover bugs, glitches, quirks and inconsistencies.

Not your Grandpa's lint: PC-lint has introduced several spectacular and revolutionary innovations in the art of static program analysis. Taking clues from initializers, assignments, and conditionals, variable and member values are tracked, enabling reports on potential uses of null pointers and out-of-bounds subscripts.

New with Version 8: Interfunction value tracking – Actual argument values are used to initialize parameters; return values are computed; a multi-pass operation (you control the number of passes) allows you to plumb the depths of function behavior to arbitrary levels.

Plus Our Traditional C/C++ Warnings: Uninitialized variables, inherited non-virtual destructors, strong type mismatches, ill-formed macros, inadvertent name-hiding, suspicious expressions, etc., etc.

Full Language Support for ANSI/ISO C and C++.

PC-lint for C/C++ *$239*
Numerous compilers/ libraries supported. Runs on Windows, MS-DOS, and OS/2.

FlexeLint for C/C++
The same great product for other operating systems. Runs on all UNIX systems, VMS, mainframes, etc. Distributed in shrouded C source form. Call for pricing.

30 Day Money Back Guarantee

Gimpel Software

Serving the C/C++ Community for 15 Years.

3207 Hogarth Lane, Collegeville, PA 19426

CALL TODAY (610) 584-4261 Or FAX (610) 584-4266

www.gimpel.com

PA add 6% sales tax.

PC-lint and FlexeLint are trademarks of Gimpel Software

MIPSpro7.3.1.2m and Visual C++ 6.0 failed one of the two test cases. The bars for Clause 4 might indicate that the latter two compilers did poorly on this clause but, in fact, they failed only a single test case.

One of our goals was to measure the progress of the GNU C++ compiler toward ISO conformance. Figure 3 reveals that gcc is making steady progress toward conformance. The graph contains three bars for each of the clauses, where the top bar for a clause represents the most recent version of gcc in our tests (gcc 3.0.4), the second bar represents gcc 2.96, and the third bar represents the oldest version (gcc 2.95.2). For all clauses we tested, gcc 3.0.4 performed as well or better than gcc 2.95.2. Also, for Clause 14, which tests templates, gcc has shown steady improvement toward ISO conformance.

Conclusion

We have described the construction of a Python test framework that lets us use the same test harness for compilers on different platforms. We have used the examples from the ISO Standard together with the described outcomes to construct test cases to measure the conformance of popular compilers. Since nomenclature for include files varies across vendors, we conditionally compile the correct header file for the respective compiler. Our results indicate that all of the compilers in our test suite performed very well and that the GNU C++ compiler is moving steadily toward conformance to the ISO Standard. We believe that our approach is adaptable to other forms of testing where cross-platform compatibility is important. We are currently extending our framework to perform unit testing on C++ classes.

DDJ

Compiler	3	4	5	6	7	8	9	10	11	12	13	14	15	Failures	%Passed
🟫gcc 3.0.4	8	0	1	2	9	9	1	0	12	3	0	24	1	70	90.7
🟪MIPSpro 7.3.1.2m	13	1	0	2	8	4	2	0	7	3	3	29	0	72	90.5
gcc 2.96	8	0	2	2	10	8	1	0	12	3	2	27	1	76	90.0
gcc 2.95.2	8	0	1	2	14	10	1	1	12	3	2	31	2	86	88.6
🟫Borland 5.5.1	13	0	1	0	16	7	4	1	11	10	2	38	1	104	86.4
🟥Visual C++ 6.0	19	1	4	9	17	15	7	6	9	13	19	80	2	201	73.6
Total Cases	**88**	**2**	**20**	**14**	**84**	**90**	**38**	**41**	**54**	**53**	**52**	**217**	**7**	**760**	**—**

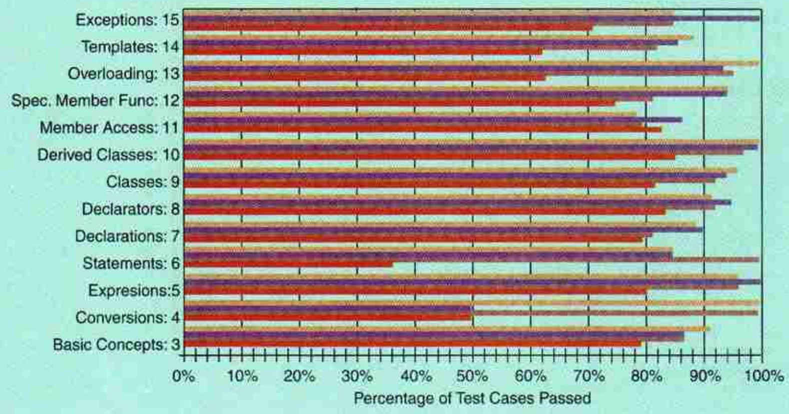


Figure 2: *Results of our study.*

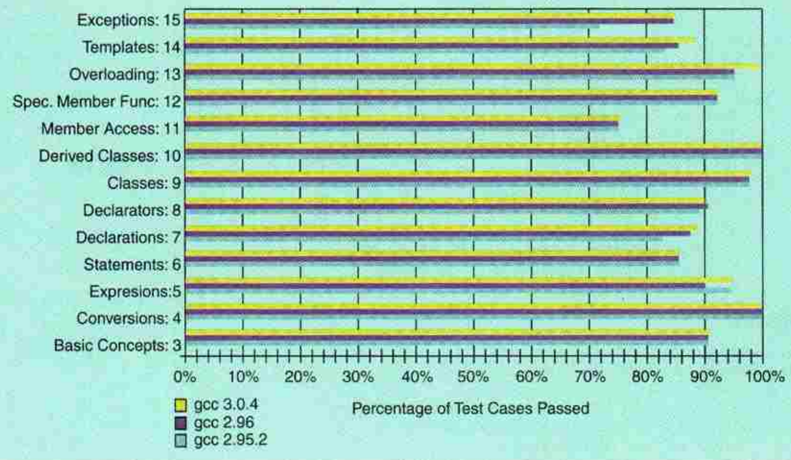


Figure 3: *Progression of gcc toward conformance.*

Listing One

```
class BinaryError( Exception): pass
class InvalidBinar yError(BinaryE rror): pass
class Binary:
   def _init_ ( self, n = 'Ø' ):
      """number s are stored as integers"""
      if int(n) < Ø:
          raise InvalidBinary Error, n
          "Negative numbers are invalid"
      self.number = int( n )
   def _add_ ( self, rhs ):
      return Binary(self.number+rhs.number)
   def _mul_ ( self, rhs ):
      return Binary(self.number*rhs.number)
   def _eq_ ( self, rhs ):
      return (self.num ber==rhs.numbe r)
   def _ne_ ( self, rhs ):
      return (self.num ber!=rhs.numbe r)
   def _str_ ( self ):
      """Prints the number in binary format"""
      number = self.number
      result = []
      while number:
          result.inser t(Ø, str(number%2))
          number = number / 2
      return "".join(result)
def test():
   bin1 = Binary( 17 )
   bin2 = Binary( 127 )
   print bin1, " + ", bin2, " is ", bin1+bin2
   print bin1, " * ", bin2, " is ", bin1*bin2
   if bin1 != bin2: print "Not Equal"
   else : print "Equal"
   if bin2 != Binary(127): print "Not Equal"
   else : print "Equal"
if name == " _main_":
   test()
```

Listing Two

```
import unittest
from binary import Binary
from binary import InvalidBinaryError
class BinaryTest(unittest.TestCase):
   def setUp(s elf):
       self.n = Binary(Ø)
   def tearDow n(self): pass
   def testZero(self):
       self.asse rtEqual(self.n, Binary(Ø))
    def testAdd ition(self):
       rhs = Binary(7)
       self.assertEqual((self.n+rhs), Binary(7))
       self.assertEqual((Binary(7)+rhs), Binary(14))
   def testMultiplication(self):
       for n in range(1ØØ):
           self.assertEqual(Binary(n)*Binary(n), \
           Binary(n*n))
```

(continued on page 78)

(continued from page 77)

```
class BadInputTest(unittest.TestCase):
  def setUp(self):
      self.n = Binary(0)
  def testNegative(self):
      """Binary should fail with negative input"""
      self.assertRaises(InvalidBinaryError, \
      Binary, -1)
  def testDecimal(self):
      self.assertRaises(InvalidBinaryError,\
      Binary, 0.5)
if name == " _main_ ":
  unittest.main ()
```

Listing Three

```
#!/usr/bin/env python2.2
import unittest, fnmatch, os, sys, cpptests
def doTests(fu llpath, directory):
    dirlist = os.listdir(fullpath)
    runner = unittest.TextTestRunner ()
    suite = unittest.TestSuite()
    for fname in dirlist :
       if os.path.isfile(fullpath+' /'+fname) \
           and fnmatch.fnma tch(fname, "*.cpp"):
           gen = cpptests.CppTestCase("test Execute", \
               fname[:-4 ])
           suite.addTest( gen )
    runner.run(suite)
def cleanUp(fu llpath):
     dirlist = os.listdir(fullpath)
     for fname in dirlist :
        if os.path.isfile(fullpath+' /'+fname) and \
           (fnmatch.fnmatch(fname, "*.o")
           or fnmatch.fnmatch(fname, "*.obj")
           or fnmatch.fnmatch(fname, "*.exe")):
           os.remove(fname)
if __name__ == "__main__":
     if len(sys.argv) != 2:
         print "usage: ", sys.argv[0], " <clause dir>"
     else :
         directory = sys.argv[1]
         fullpath = os.getcwd() +'/'+directory
         if os.path.isdir(fullpath):
             os.chdir(fullpath)
             doTests(fullpath, directory)
             cleanUp(fullpath)
         else :
             print directory, " is not in this directory"
             print "Current directory is: ", os.getcwd()
```

Listing Four

```
import unittest, os, re
class CppTestCase(unittest.TestCase):
    def __init__ (self, testfun, fname):
       unittest.TestCase.init (self, testfun)
       self.compile = [ "g++ -c -DGCC29x %s.cpp", \
          "g++ -Wno- -c -DGCC30x %s.cpp", \
          "cl /w /nologo /c -DMSVC6x %s.cpp", \
          "bcc32 -w- -q -c -DBORLAND 55 %s.cpp", \
          "CC -c -DMIPS %s.cpp ]
       self.link = [ "g++ -o %s.exe %s.o", \
          "g++ -o %s.exe %s.o", \
          "cl /nologo /w /Fe%s.exe %s.obj", \
          "bcc32 -q -e%s.exe %s.obj" , \
          "CC -o %s.exe %s.o" \
          ]
       self.fileName = fname
       self.toPass = not (fname[:4] == "fail")
       self.hasMain = 0
       self.directory = os.getcwd()
    def setUp(self):
       print "Executing: %s.cpp" % self.fileName
       oldFile = open(self.file Name+".cpp", "r")
       currentline = oldFile.readline()
       while currentline :
          if re.search("main", currentline):
             self.hasMain = 1
             break;
           current line = oldFile.readline()
       oldFile.close()
    def tearDown(self): pass
    def testExecute(self):
    #Code for this method in Listing Five
```

Listing Five

```
def testExecute(self):
   def testExecute(self):
      executed = 0
      compiled = (os.system(self.compile[0] % \
         self.fileName) == 0)
      if compiled and self.hasMain:
         linked = (os.system(self.link[0] % \
           (self.fileName, self.fileName)) == 0)
         if linked:
           executed = (os.system ("%s.exe" % \
               self.fileName) == 0)
      if self.toPass and self.hasMain \
         and compiled and executed:
         print "PASS: Semantics properly supported"
      elif self.toPass and self.hasMain \
         and compiled and not executed:
         print "FAIL: Semantics not supported"
         failures.add(self.fileName)
      elif self.toPass and self.hasMain and \
         not compiled and not executed:
         print "FAIL: Should have compiled"
         failures.add(self.fileName)
      elif self.toPass and not self.hasMain and \
         compiled and not executed:
         print "PASS: Compiled as expected"
      elif self.toPass and not self.hasMain and \
         not compiled and not executed:
         print "FAIL: Should have compiled"
         failures.add(self.fileName)
      elif not self.toPass and self.hasMain \
         and compiled and executed:
         print "FAIL: Executed but shouldn't have"
         failures.add(self.fileName)
      elif not self.toPass and self.hasMain \
         and compiled and not executed:
         print "PASS: Semantics properly supported"
      elif not self.toPass and self.hasMain \
         and not compiled and not executed:
         print "PASS: Did not compile, as expected"
      elif not self.toPass and not self.hasMain \
         and compiled and not executed:
         print "FAIL: Should not have compiled "
         failures.add(self.fileName)
      elif not self.toPass and not self.hasMain \
         and not compiled and not executed:
         print "PASS: Did not compile, as expected"
      else:
         print "logic errors"
```

Listing Six

```
namespace N {
  int i;
  int g( int a) {return a;}
  int j();
  void q();
}
namespace { int l = 1; }
namespace N {
  int g( char a) { // overloads N::g( int )
     return l+a;   // l is from unnames namespace
  }
  int i;            // error: duplicate definition
  int j()           // OK: duplicate function declaration
  int j()  {        // OK: definition of N::j()
     return g(i);  // calls N::g( int )
  }
  int q();           // error: different return type
}
```

Listing Seven

```
#if defined( BORLAND55)
#include <mem.h>
#else
#include < memory.h >
#endif
struct T { int a; };
#define N sizeof(T)
int main () {
   char buf[N];
   T obj;
   obj.a = 1138;
   memcpy(buf, &obj, N);
   memcpy(&obj, buf, N);
   if (1138 != obj.a) return 1;
   return 0;
}
```

Listing Eight

```
typedef int f;
struct A {
  friend void f(A &);
  operator int ();
  void g(A a) {
    f(a);
  }
} ;
```

DDJ