

Testing C++ Compilers for ISO Language Conformance

Brian A. Malloy, Scott A. Linde, Edward B. Duffy
Computer Science Department
Clemson University
Clemson, SC 29634
USA.

{malloy,slinde,eduffy}@cs.clemson.edu

James F. Power
Computer Science Department
National University of Ireland
Maynooth, Co. Kildare
Ireland

James.Power@may.ie

Abstract

In this paper, we describe our construction of a test harness to measure conformance of some popular C++ compilers and to measure the progress of the gcc C++ compiler as it moves toward ISO conformance. In an attempt to apply the same standard to all of the vendors, we use the same test cases and the same 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 the Python language provided the functionality that we needed with its scripting facility, its platform independence and its object orientation to facilitate code reuse. Python includes a testing framework as a module of the language and we have extended the framework to measure C++ ISO conformance.

1 Introduction

Conformance to a standard is one of the most important endeavors that a compiler vendor might undertake. Conformance can affect acceptance of the compiler and, in many cases, can have an impact on the language itself. Conformance enables code portability and wider use of the language and corresponding libraries. Even if code portability is not important, conformance facilitates documentation so that text books and language reference manuals have a common frame of reference.

Conformance is especially important and difficult for C++ because the language standard was slow to develop and acceptance of the standard occurred 14 years after the introduction of the language. By 1998, when the ISO standard was accepted[1], there were many established and accepted C++ compilers in common use.

Reference [6] 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. We overview this roundup in Section 5. However, we found the roundup to be inconclusive. We were interested in measuring conformance of the commonly used C++ compilers in an unbiased and conclusive fashion. Moreover, we wanted to investigate the GNU compiler suite to determine if gcc versions were moving toward conformance.

In this paper, we describe our construction of a test harness to measure conformance of some popular C++ compilers and to measure the progress of the gcc C++ compiler as it moves toward ISO conformance. In an attempt to apply the same standard to all of the vendors under consideration, we use the same test cases and the same 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 the Python language provided the functionality that we needed with its scripting facility, its platform independence and its object orientation to facilitate code reuse. Moreover, unlike any other language, Python includes a testing framework as a module of the language. This testing framework is a Python module, *unittest*, written by Purcell[5], and patterned after the JUnit framework developed by Beck and Gamma[2], and included with Python versions 2.1 and later[7].

We have extended the framework to facilitate measurement of ISO conformance. We use our extended framework in all test case executions. In an attempt to avoid bias for or against any vendor, our test case selection is based on test cases found directly in the ISO C++ standard[1]. All of our test cases are actual examples listed specifically in the standard with outcomes specified.

In the next section, we provide background about *unittest*, and in Section 3, we describe the construction of our Python test harness. We assume that the reader is famil-

```

1 class BinaryError(Exception): pass
2 class InvalidBinaryError(BinaryError): pass

3 class Binary:
4     def __init__( self, n = '0' ):
5         """numbers are stored as integers"""
6         if int(n) < 0:
7             raise InvalidBinaryError,\
8                 "Negative numbers are invalid"
9         self.number = int( n )
10    def __add__( self, rhs ):
11        return Binary(self.number+rhs.number)
12    def __mul__( self, rhs ):
13        return Binary(self.number*rhs.number)
14    def __eq__( self, rhs ):
15        return (self.number==rhs.number)
16    def __ne__( self, rhs ):
17        return (self.number!=rhs.number)
18    def __str__( self ):
19        """Prints the number in binary format"""
20        number = self.number
21        result = []
22        while number:
23            result.insert(0, str(number%2))
24            number = number / 2
25        return "".join(result)

26 def test():
27     bin1 = Binary( 17 )
28     bin2 = Binary( 127 )
29     print bin1, " + ", bin2, " is ", bin1+bin2
30     print bin1, " * ", bin2, " is ", bin1*bin2
31     if bin1 != bin2: print "Not Equal"
32     else: print "Equal"
33     if bin2 != Binary(127): print "Not Equal"
34     else: print "Equal"

35 if __name__ == "__main__":
36     test()

```

Figure 1. Binary number class and test code.

iar with Python. The Python newbie can find gentle introductions to Python at *Python.org*. For those who are well acquainted with Python, we highly recommend references [3] and [4]; these are excellent resources, but only for those well acquainted with Python and object technology. In Section 4 we describe our test case extraction and in Section 5 we overview the C++ roundup that appeared in reference [6]. In Section 6 we present a case study where we apply our framework to some popular compilers running on several different platforms. We measure the performance of the compilers on our test suite, including a discussion of a difficult-to-parse test case from clause three of the standard. We draw conclusions in Section 7.

2 The unittest Testing Framework

In this section, we overview *unittest*, the unit testing framework included in Python versions 2.1 and later. We first illustrate a Python class, *Binary*, and show how the class might be tested without the *unittest* module. We then present class *BinaryTest* and illustrate its use and advantages in testing *Binary*.

2.1 Incorporating test cases into the module

In Python, each class begins with the keyword **class** and each function begins with the keyword **def**. There are classes defined on lines 1, 2 and 3 of Figure 1 and functions defined on lines 4, 10, 12, 14, 16, 18 and 26 of the figure. The classes on lines 1 and 2 do not contain any data or methods, as indicated by the keyword **pass**: they are used to handle exceptions. Class *Binary*, lines 3 to 25 in Figure 1, represents an abstraction for binary numbers. Users of the class instantiate *Binary* numbers using positive decimal numbers and 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 to 9, is the constructor for *Binary*. Functions *__add__*, *__mul__*, *__eq__*, and *__ne__*, overload the +, *, ==, and != operators for two binary numbers. Function *__str__*, lines 18 to 25, enable output of binary numbers, similar to *operator <<* for C++, and *toString* for Java.

Function *test()*, lines 26 to 34, represents one approach to testing class *Binary*. Binary numbers *bin1* and *bin2* are instantiated on lines 27 and 28, and then used in addition, multiplication and comparison on lines 29 to 34. Function *test()* will be invoked on line 36 when the module is executed in stand-alone fashion.

These tests are not intended to be exhaustive, but rather they illustrate one approach to testing a module. However there are drawbacks in using this approach to testing. First, the code to test the module is included in the module itself, which can distract the reader 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 that the output 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.

2.2 Using unittest to separate testing

Figure 2 illustrates an alternative, using the *unittest* module, to the testing approach shown in Figure 1. The PyUnit home page can provide more information about the *unittest* framework[5] and the *Python Library Reference*

```

1 import unittest
2 from binary import Binary
3 from binary import InvalidBinaryError

4 class BinaryTest(unittest.TestCase):
5     def setUp(self):
6         self.n = Binary(0)
7     def tearDown(self): pass
8     def testZero(self):
9         self.assertEqual(self.n, Binary(0))
10    def testAddition(self):
11        rhs = Binary(7)
12        self.assertEqual((self.n+rhs), Binary(7))
13        self.assertEqual((Binary(7)+rhs), Binary(14))
14    def testMultiplication(self):
15        for n in range(100):
16            self.assertEqual(Binary(n)*Binary(n),\
17                Binary(n*n))

18 class BadInputTest(unittest.TestCase):
19     def setUp(self):
20         self.n = Binary(0)
21     def testNegative(self):
22         """Binary should fail with nega-
23         tive input"""
24         self.assertRaises(InvalidBinaryError,\
25             Binary, -1)
26     def testDecimal(self):
27         self.assertRaises(InvalidBinaryError,\
28             Binary, 0.5)

29 if __name__ == "__main__":
30     unittest.main()

```

Figure 2. Using *unittest*. This figure illustrates a better approach to testing a Python module using the module *unittest*. In this example, the methods that are prefixed with “test” are test cases that can be used to test class *Binary*.

summarizes the *unittest* module[7]. Mark Pilgrim’s excellent public-domain book, *Dive Into Python*, presents in-depth examples of using *unittest*, including a thorough test suite for a Roman numeral module[4]

The example in Figure 2 imports three modules, *unittest*, *Binary* and *InvalidBinaryError* on lines 1, 2 and 3 respectively. The *import* format on line 1 requires that all uses of the imported module be prefixed with the module name; the format used on lines 2 and 3 do not require the module name prefix. The two classes in Figure 2, *BinaryTest* on line 4, and *BadInputTest* on line 18, encapsulate the test cases that we will use to test class *Binary*. Both of these 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 on lines 4 and 18. By inheriting from *TestClass*, we acquire useful methods to facilitate testing and we exploit many of them in this example.

The two classes, *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* on lines 5 and 6, instantiates a *Binary* zero as a data member. Method *tearDown* on 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 test case contains *assertEqual* statements to determine if the values returned by the *Binary* API is correct. The statement on 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 they are not equal *assertEqual* will raise an exception and the test will fail. *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.

The *BadInputTest* class uses the *assertRaises* statement to make sure that the *Binary* API handles bad input. The *testNegative* test case on line 21 of Figure 2 makes sure that *Binary* raises the exception *InvalidBinaryError*, line 23, if the user of the API attempts to instantiate a negative binary number; this test case passes. However, *testDecimal*, the method to test for decimal input on line 25 does not pass since the *Binary* API does not raise an exception when the user attempts to instantiate a decimal *Binary*. This is the only test case in Figure 2 that fails.

unittest will automatically call *setUp* and *tearDown* before and after each test case execution and the three test methods in *BinaryTest* will be invoked automatically by *unittest*. Similarly, the *setUp* routine for *BadInputTest* and both of its negative test cases will be called automatically.

In fact, there is a lot that will happen automatically when we use the example in Figure 2 in stand-alone fashion. When *main* is invoked on 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 will then be 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 function *cmp*. The testing framework provides an environment in which the test cases can execute and a report is generated. Alternatively, the user can construct the test suite and pass the name of the test method as a parameter to the newly constructed test suite; we use this technique in Section 3.

Executing the tests in Figure 2, produces the output below, with 5 test cases executed and one test case failure (*testDecimal*):

```

1 #!/usr/bin/env python2.2
2 import unittest, fnmatch, os, sys, cpptests

3 def doTests(fullpath, directory):
4     dirlist = os.listdir(fullpath)
5     runner = unittest.TextTestRunner()
6     suite = unittest.TestSuite()
7     for fname in dirlist:
8         if os.path.isfile(fullpath+'/'+fname)\
9             and fnmatch.fnmatch(fname, "*.cpp"):
10            gen = cpptests.CppTestCase("testExecute",\
11                fname[:-4])
12            suite.addTest( gen )
13    runner.run(suite)

14 def cleanUp(fullpath):
15     dirlist = os.listdir(fullpath)
16     for fname in dirlist:
17         if os.path.isfile(fullpath+'/'+fname) and\
18            (fnmatch.fnmatch(fname, "*.o")
19             or fnmatch.fnmatch(fname, "*.obj")
20             or fnmatch.fnmatch(fname, "*.exe")):
21            os.remove(fname)

22 if __name__ == "__main__":
23     if len(sys.argv) != 2:
24         print "usage: ", sys.argv[0], " <clause dir>"
25     else:
26         directory = sys.argv[1]
27         fullpath = os.getcwd()+'/'+directory
28         if os.path.isdir(fullpath):
29             os.chdir(fullpath)
30             doTests(fullpath, directory)
31             cleanUp(fullpath)
32     else:
33         print directory, " is not in this directory"
34     print "Current directory is: ", os.getcwd()

```

Figure 3. *The Test Case Generator.*

```

1 import unittest, os, re

2 class CppTestCase(unittest.TestCase):
3     def __init__(self, testfun, fname):
4         unittest.TestCase.__init__(self, testfun)
5         self.compile = [ "g++ -c -DGCC29x %s.cpp", \
6             "g++ -Wno- -c -DGCC30x %s.cpp", \
7             "cl /w /nologo /c -DMSVC6x %s.cpp", \
8             "bcc32 -w- -q -c -DBORLAND55 %s.cpp", \
9             "CC -c -DMIPS %s.cpp" \
10        ]
11        self.link = [ "g++ -o %s.exe %s.o", \
12            "g++ -o %s.exe %s.o", \
13            "cl /nologo /w /Fe%s.exe %s.obj", \
14            "bcc32 -q -e%s.exe %s.obj", \
15            "CC -o %s.exe %s.o" \
16        ]
17        self.fileName = fname
18        self.toPass = not (fname[:4] == "fail")
19        self.hasMain = 0
20        self.directory = os.getcwd()

21    def setUp(self):
22        print "Executing: %s.cpp" % self.fileName
23        oldFile = open(self.fileName+".cpp", "r")
24        currentline = oldFile.readline()
25        while currentline:
26            if re.search("main", currentline):
27                self.hasMain = 1
28                break;
29            currentline = oldFile.readline()
30        oldFile.close()

31    def tearDown(self): pass

32    def testExecute(self):
33        #Code for this method in Figure 5

```

Figure 4. *The Test Case Class.*

```

% 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)

```

3 Construction of the Test Harness

In this section we present the class, *CppTestCase*, that we derive from class *TestCase*, in the *unittest* module. We use *CppTestCase* as a wrapper for our C++ test cases that test

compiler conformance. We begin the section by presenting a test case generator.

3.1 Test Case Generation

Figure 3 illustrates our test case generation module that we call *runtests.py*, consisting of two functions, *doTests* on lines 3 to 13, and *cleanUp* on lines 14 to 21. Each Python module contains a global namespace with an identifier called *__name__* that stores the modules 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 on line 22 evaluates to *true*.

When the session begins, the *if* statement running from lines 23 to 34 verifies the 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 on 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, functions `doTests` and `cleanUp` are called to do the testing of the clause and clean up afterward.

Function `doTests` accepts two arguments (line 3): 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, instantiates a test runner, using `TextTestRunner`, and a test suite, using `TestSuite`. Classes `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 that it has the proper extension (our test cases have a `.cpp` extension, but there may be other files in the directory). 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` is to tell the `TextTestRunner` object, `runner`, to run the tests, line 13.

Function `cleanUp`, lines 14 to 21 of Figure 3, cleans up after the test suite is executed. We could have used the `tearDown` method in `CppTestCase`, derived from `unittest` to clean up after each individual test case, but we found it more efficient to simply 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.

3.2 The C++ Test Case Wrapper

Figures 4 and 5 illustrate class `CppTestCase`, a wrapper for our C++ test cases extracted from the ISO C++ standard. The class contains four methods. Method `init`, lines 3 to 18, is the class constructor, and `setUp`, lines 19 to 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 described in the previous section. Function `testExecute` is the longest method in the class and we show only the signature in Figure 4; the code for `testExecute` is in Figure 5.

The constructor for `CppTestCase`, `init` in Figure 4, initializes the data used in the test case. The method begins by calling the constructor of the super class, `TestCase`, on line 4, passing the name of the function that will execute the test case; in our framework this function is `testExecute`, introduced earlier in our discussion of Figure 3. It is im-

```

1 def testExecute(self):
2 def testExecute(self):
3     executed = 0
4     compiled = (os.system(self.compile[0] % \
5         self.fileName) == 0)
6     if compiled and self.hasMain:
7         linked = (os.system(self.link[0] % \
8             (self.fileName, self.fileName)) == 0)
9         if linked:
10            executed = (os.system("%s.exe" % \
11                self.fileName) == 0)
12
13 if self.toPass and self.hasMain \
14     and compiled and executed:
15     print "PASS: Semantics properly supported"
16 elif self.toPass and self.hasMain \
17     and compiled and not executed:
18     print "FAIL: Semantics not supported"
19     failures.add(self.fileName)
20 elif self.toPass and self.hasMain and \
21     not compiled and not executed:
22     print "FAIL: Should have compiled"
23     failures.add(self.fileName)
24 elif self.toPass and not self.hasMain and \
25     compiled and not executed:
26     print "PASS: Compiled as expected"
27 elif self.toPass and not self.hasMain and \
28     not compiled and not executed:
29     print "FAIL: Should have compiled"
30     failures.add(self.fileName)
31 elif not self.toPass and self.hasMain \
32     and compiled and executed:
33     print "FAIL: Executed but shouldn't have"
34     failures.add(self.fileName)
35 elif not self.toPass and self.hasMain \
36     and compiled and not executed:
37     print "PASS: Semantics properly supported"
38 elif not self.toPass and self.hasMain \
39     and not compiled and not executed:
40     print "PASS: Did not compile, as expected"
41 elif not self.toPass and not self.hasMain \
42     and compiled and not executed:
43     print "FAIL: Should not have compiled"
44     failures.add(self.fileName)
45 elif not self.toPass and not self.hasMain \
46     and not compiled and not executed:
47     print "PASS: Did not compile, as expected"
48 else:
49     print "logic errors"

```

Figure 5. The `testExecute` method for the `Test Case Class`.

portant that the *CppTestCase* constructor explicitly call the constructor of the superclass because, in Python, constructors for superclasses are not automatically invoked. Lines 5 through 14 initialize a list that contains the calls for each of the compilers that 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 *testExecute* to determine the name of the files to include; we discuss our solution to the variation in include file names in the next section. 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 to 28 in Figure 4, parses the test case to determine if it contains a function main. If it does, the *hasMain* flag is set to true on line 20 and the test case will be compiled, linked and executed.

The code for the final method of *CppTestCase*, *testExecute*, is shown in Figure 4. 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 Figure 4 where the results are assigned to variables *compiled*, *linked* and *executed*. All of the systems that we used follow the convention that upon successful compile, link or execute, a zero value is returned, otherwise a non-zero value is returned. However, the Windows 95 and 98 operating systems do not follow this convention, but rather return zero value for both success and failure. Thus, our framework will not provide correct results on these two systems.

Some examples in the ISO standard are intended to compile, others are intended 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 or not 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-six percent of the test cases that we extracted from the standard are negative test cases and these form an important part of our measurement of ISO conformance.

The most important function of *testExecute* is to determine whether or not the test case passes or fails, based on the values of the two 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 and if the *hasMain* flag is true than this test case is supposed to link, execute and possibly give a specified output. We

```

1 namespace N {
2     int i;
3     int g(int a) { return a; }
4     int j();
5     void q();
6 }
7 namespace { int l = 1; }
8 namespace N {
9     int g(char a) { // overloads N::g(int)
10        return l+a; // l is from unnames namespace
11    }
12    int i; // error: duplicate definition
13    int j() // OK: duplicate function declaration
14    int j() { // OK: definition of N::j()
15        return g(i); // calls N::g(int)
16    }
17    int q(); // error: different return type
18 }

```

Figure 6. *Namespace example.*

make a judgement about whether or not 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, 6 of which are infeasible. We model the ten possible outcomes on lines 12 through 48 of Figure 4.

4 Test Case Extraction

Each clause in the standard contains C++ examples and descriptions of the expected outcome. In section 5 we discuss the discrepancy in the number of test cases extracted from the ISO standard by different testers. Given the complexity of the examples and the inclusion of multiple errors in a single example, we are not surprised by this discrepancy.

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 the example in Figure 6, taken from clause 3 of the ISO standard, which specifies rules for name lookup in namespaces. The code in the figure 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 of the example; 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 Figure 6: 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

```

1  #if defined(BORLAND55)
2  #include <mem.h>
3  #else
4  #include <memory.h>
5  #endif

6  struct T { int a; };
7  #define N sizeof(T)
8  int main () {
9      char buf[N];
10     T obj;
11     obj.a = 1138;
12     memcpy(buf, &obj, N);
13     memcpy(&obj, buf, N);
14     if (1138 != obj.a) return 1;
15     return 0;
16 }

```

Figure 7. Handling include file variations.

testing approach might generate many more test cases than three using the example in Figure 6.

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 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 *string s*; might be modified to *int s*; 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. The example in Figure 7, taken from clause three of the ISO standard, illustrates a test case where the function *memcpy* is part of the test. In the figure, *memcpy* is used to copy a value from a **struct** to a buffer and 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 through 5 of Figure 7, chooses the file to include based on the compiler under test.

5 The C++ Conformance Roundup

Reference [6] 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 major suppliers of C++ conformance test suites were asked to evaluate the compilers; these were: *Dinkumware Ltd.*, *Perennial Inc.* and *Plum Hall Inc.*. The compilers being tested for conformance included IBM, Sun, Kuck and Associates, Metrowerks, Intel, HP, MSFT/Ze,

GCC, BorC++, BCC, Comeau and MSFT/Ze.

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 4,356 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 from the standard with outcomes specified. Using this approach we extracted 760 test cases.

6 Case Study

```

1  typedef int f;
2  struct A {
3      friend void f(A &);
4      operator int();
5      void g(A a) {
6          f(a);
7      }
8  };

```

Figure 9. A Test case that all compilers failed.

In this section, we apply our testing framework to some popular C++ compilers running on several different platforms. The compilers in our study include *Borland 5.5.1*, *Visual C++ 6.0*, *gcc 2.95.2*, *gcc 2.96*, *gcc 3.0.4* and *MIP-Spro7.3.1.2m*. We executed the test cases for *Borland 5.5.1* and *Visual C++ 6.0* on Windows NT and Windows 2000 platforms; all of the rest of the test cases were executed on Linux or Solaris systems running version 7.1 of Red Hat or Solaris SunOS version 5.8. We have tested the framework on Python versions 1.5 through 2.2; we note that 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.

The table in Figure 8 summarizes our results, where the first column lists the names of the compilers and the columns labeled 3 through 15 list the results for clauses 3 through 15 for the respective compilers. The column labeled *Failures* lists the total number of test cases failed by

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	0	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
VisualC++ 6.0	19	1	4	9	17	15	7	6	9	13	19	80	2	201	73.6
VisualStudio 7.0	15	0	4	4	15	15	7	4	11	12	18	67	2	174	77.1
Total Cases	88	2	20	14	84	90	38	41	54	53	52	217	7	760	—

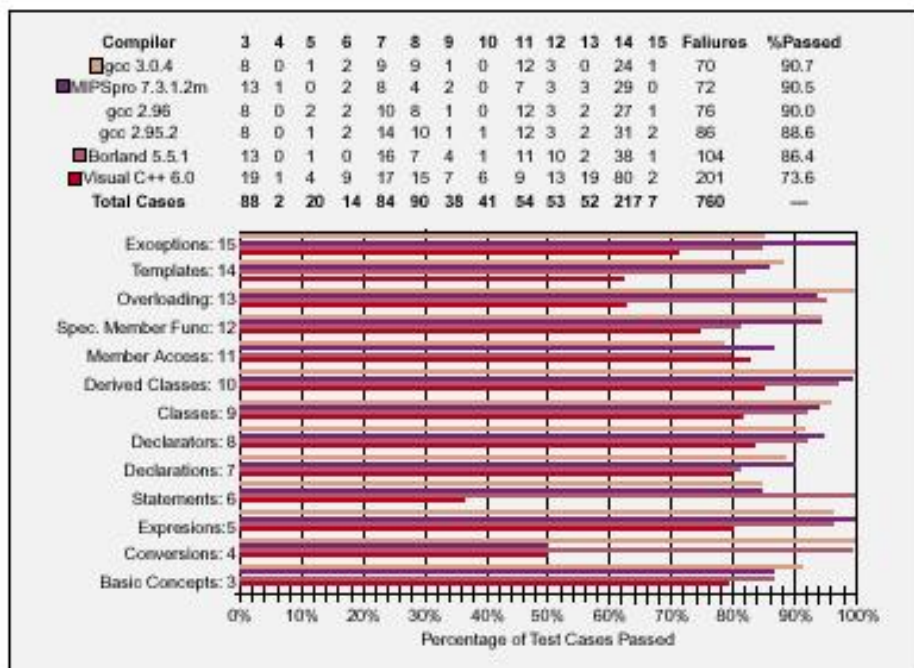


Figure 8. Results of our study.

the respective compiler and the final column, *% Passed*, represents the percentage of test cases that passed. The bottom row of the table in Figure 8 lists the number of test cases in each of the respective clauses, with the total number of test cases at 760. For example, column one of the table 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 8 shows that the first three compilers passed at least 90% of the test cases, with the *gcc 2.95.2* and *Borland* compilers very close to 90%. Moreover the *VisualC++ 6.0* compiler also performed quite well in the tests.

Our goal here is to show that our testing framework is extensible to multiple platforms and to provide some measure of how the compilers stack up against the examples in the ISO standard. We are not considering compile speed, efficiency of the optimized code or the friendliness of the environments of the respective compilers. 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 Figure 9, a test case that all compilers failed. The purpose of the test case is to illustrate that the expression on line 6 of the example is not a function call and that argument-dependent name lookup does not apply; rather the expression is a cast equivalent to *int(a)[1, §3.4.1]*. However, the compilers that we tested find the expression on line 6 to be a redeclaration of the friend function on line 3 and become confused with the **typedef** on line 1. None of the compilers were able to compile this example correctly.

The graph in Figure 8 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 each of four compilers: *gcc 3.0.4*, *MIPSpro 7.3.1.2m*, *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 *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 gcc C++ compiler toward ISO conformance and Figure 10 reveals that gcc is making steady progress toward conformance. The graph in the figure 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 that 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.

7 Concluding Remarks

We have described the construction of a Python test framework that allows us to 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.

8 Acknowledgement

We would like to thank Stephen Alfors for his construction of the web page. The Python code together with clause 3 test cases from the ISO standard are available for download at our web site at the following URL:

www.cs.clemson.edu/~malloy/projects/ddj

References

- [1] *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, first edition, September 1998. ISO/IEC JTC 1.
- [2] E. Gamma and K. Beck. Test infected: Programmers love writing tests. Using JUnit to automatically generate test cases, <http://members.pingnet.ch/gamma/junit.htm> 2001.
- [3] M. Lutz. *Programming Python*. O'Reilly, 2nd edition, 2001.
- [4] M. Pilgrim. *Dive Into Python*. Freely available at <http://diveintopython.org/>, 2002.

[5] S. Purcell. Python unit testing framework. *documentation freely available at <http://pyunit.sourceforge.net/>*, March 2002.

[6] H. Sutter. C++ conformance roundup. *C/C++ Users Journal*, 19(4), April 2001.

[7] Guido van Rossum. *Python Library Reference*. Python Software Foundation, 2001.

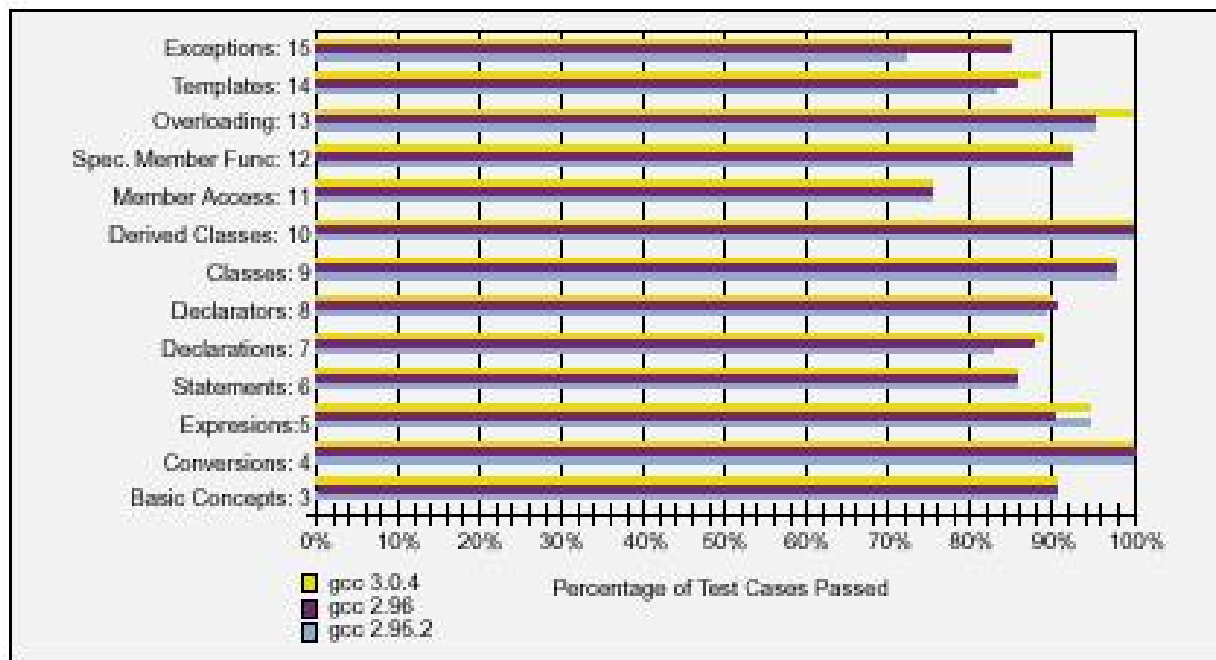


Figure 10. *Progression of gcc toward conformance.*