

Progression Toward Conformance for C++ Language Compilers

Tanton H. Gibbs
Software Development
Acxiom Corporation
Conway, AR 72032
tanton@deltafarms.com

Brian A. Malloy
Computer Science Dept.
Clemson University
Clemson, SC 29634, USA
malloy@cs.clemson.edu

James F. Power
Computer Science Dept.
National University of Ireland
Maynooth, Co. Kildare, Ireland
jpower@cs.may.ie

Abstract

Establishing the conformance of a compiler is a difficult matter, especially for C++, since a standard for the language was slow to develop and acceptance of the standard occurred years after the introduction of the language. In this paper, we re-visit our conformance study presented in June 2002. We describe the construction of our test suite composed of test cases extracted from the code examples in the ISO C++ standard together with the process that we use to remove test cases extracted from examples under investigation by the working committee. We then use our test suite to provide some measure of conformance to the ISO standard for eight popular C++ compilers.

1 Introduction

Conformance to a standard is becoming recognized as one of the most important assurances compiler vendors can provide to users of the language. Conformance enables code portability and wider use of the language and corresponding libraries. However, establishing the conformance of a compiler is a difficult matter, especially for C++, since a standard for the language was slow to develop and acceptance of the standard occurred years after the introduction of the language. Many of the most popular C++ compilers developed with the language so that the compiler vendors feel obligated to enable compilation of legacy code written before the C++ language standard was ratified.

One possible approach toward measuring the conformance of a compiler to a standard is to construct a suite of test cases that measure either the acceptance of correct code or the rejection of incorrect

code. However, the construction of a test suite is difficult since there is no central repository of test cases that conform to the standard. One approach might suggest the construction of test cases from coding examples listed on web pages. However, these coding examples are almost certainly designed to work with a specific compiler and are therefore biased toward this particular compiler, which may or may not be conformant to the standard. For example, open source software is likely to compile under the *gcc* compiler, which may not be conformant. Also, older code is unlikely to exercise language constructs that have been recently introduced, such as templates or namespaces.

An alternative approach to gathering test cases from code examples on web pages is to extract them from the examples in the ISO C++ standard [3]. However, there are difficulties with this approach. One difficulty is that the examples are intended to explain or demonstrate intricate language features and are not intended to be exhaustive. Thus, some language features may go untested. A second difficulty is that the examples are not evenly distributed among the language features and some features receive more example focus than others. The template construct seems to have attracted special attention in the standard so that a compiler that does not handle a particular template construct may fail a disproportionately higher number of test cases than a compiler that can handle the construct.

However, perhaps the most daunting difficulty in using the examples in the standard to construct a test suite is that the standard itself is a work in progress. Since its ratification by the ISO committee in September of 1998, 411 core language issues have been raised by the C++ user community. Of these 411 issues, 93 have been addressed by the committee

and these changes to the standard are incorporated into *Technical Corrigendum 1* (TC1), a revision to the standard issued in 2003. Thus, the ISO C++ standard is also in a state of flux, though it is clearly moving toward a fixed point.

In this paper, we re-visit our conformance study of the C++ core language presented in June 2002 [5]. We describe the construction of our test suite composed of test cases extracted from the code examples in the ISO C++ standard together with the process that we use to remove test cases extracted from examples under investigation by the working committee. We then use our test suite to provide some measure of conformance to the ISO standard for eight popular C++ compilers. We feel that our results provide an approximate rather than a full measure of conformance since (1) the examples in the standard are not exhaustive, (2) the examples are unevenly partitioned across language constructs, and (3) the standard is a work in progress.

In the next section, we describe the process that we use to extract test cases from the standard and in Section 3 we describe the difficulties in test case extraction and the categories of issues under investigation by the working group. In Section 4 we describe changes to our Python testing framework that we presented in our previous article. In Section 5 we present our measurement of conformance for eight compilers including a measurement of the progress toward conformance for three compilers tested in our previous article. We also show the effects of compiler switches or flags on the relaxation of conformance. Finally, in Section 6 we summarize.

2 The process of extracting test cases

In this section we describe our approach for extracting test cases from examples in the ISO C++ standard. We focus only on the core language, as described in clauses 3 through 15 of the standard, and we do not consider other aspects of the standard such as the C++ preprocessor or the Standard C++ Library.

2.1 Positive and negative test cases

Each clause in the standard contains C++ examples, and most examples include descriptions of the expected behavior or outcomes if the examples are converted to programs. Other researchers and vendors have taken an approach similar to ours and we describe the discrepancy in the number of test cases

```

1 class A {
2     public:
3         static int n;
4 };
5 int main() {
6     int A;
7     A::n = 42;    // OK
8     A b;         // ill-formed: A does not name a type
9 }

```

Figure 1: *Test case extraction*. The code in this figure represents a single example in the standard but we extract two test cases from this example: one positive test case and one negative test case.

extracted from the ISO standard for the different approaches in reference [5]. Given the complexity and the inclusion of many possible outcomes for most examples, this discrepancy is not surprising.

Some of the examples illustrate code that should compile while others contain code that should not compile; we attempt to convert the former example into a *positive test case* and the latter example into a *negative test case*. In other words, if an example contains code that should not compile, the compiler will fail the corresponding test case if the test case compiles without a suitable diagnostic.¹

An example in the standard can produce many test cases. Some examples expand into multiple positive test cases while others may expand into multiple positive and negative test cases. Figure 1 illustrates a simple example taken from Section 3.4.3 of Clause 3 of the ISO standard, which specifies rules for qualified name lookup. This example is found in the first paragraph of Section 3.4.3.

Lines 1 through 4 of Figure 1 list class A with a static integer n and lines 5 through 9 list function main. In main, line 6 contains a declaration of an integer variable, A; line 7 contains an initialization of variable n in class A; and line 8 attempts to declare an instance of class A. The initialization on line 7 is valid, since A is used in a context where it is unambiguously a type name, and thus the class declaration for A is found. However, when considering the use of A in line 8, it is not clear whether the context is a declaration or an expression, and thus the default lookup rules will find the variable A on line 6, rather than the type A declared on lines 1

¹Some authors refer to positive test cases as *Conformance* tests and negative test cases as *Deviance* tests.

```

1 template <class T>
2 class complex {
3 public:
4     complex(T x) : r(x) {}
5     complex(T x, T y) : r(x), i(y) {}
6 private:
7     T r, i;
8 };

```

Figure 2: *Stub class*. We use stub classes to obviate the inclusion of non-conforming header files into the test cases.

through 4.

The code in Figure 1 represents a single example in the standard but we extract two test cases from this example: one positive test case and one negative test case. The positive test case will consist of all of the lines in Figure 1 except line 8. The negative test case will consist of the lines shown in Figure 1 except for line 7. The positive test case will pass if it compiles and executes; the negative test case will pass if it fails to compile or causes the compiler to issue a suitable diagnostic or warning. Negative test cases that compile and execute are regarded as conforming extensions to the standard provided that the compiler issues a diagnostic or warning describing the deviation from the standard.

2.2 Header files

Many of the examples in the standard, if left unaltered, will not compile. Some examples require variable or type declarations, or header file inclusions. The include library files for many compilers contain code that is non-conforming. For example, the include library files for the *gcc* compiler contain many C++ extensions that do not conform to the standard and the *gcc* documentation lists over one hundred pages of non conforming extensions². We have also found variation in nomenclature of include files across vendors.

In some cases we were able to avoid the problem of non-conforming extensions or variation in include library files 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 outcome of the test does not depend on the

²All references to *gcc* in this article refer to the C++ compiler and do not refer to the entire suite of compilers included with *gcc*.

string class. However, some test cases use member functions for classes in include files. In these cases we created stub classes and stub member functions so that no compiler was penalized because of non-standard nomenclature or extensions. A stub class or member function is a partial implementation of a class or function that simulates partial behavior of the real class or function. Figure 2 illustrates a stub class *complex*, with minimal functionality, that we used for some of the test cases from Clause 14.

3 Convergence of the standard

Our goals in choosing the ISO standard as the source of our test suite were (1) to build a test suite that covered the important issues for compilation of C++ programs, (2) to obtain test cases that were unbiased toward any particular compiler or vendor, and (3) to use the standard as an oracle to determine the validity and outcome of the test cases. We feel that our extracted test suite meets the first two goals. However, the third goal has proven more elusive. In this section we review the difficulty in judging the outcome of the test cases and we describe a test case incorrectly reported in our earlier article [5].

The examples described in the standard are intended to illustrate intricate facets of the C++ language syntax or semantics, including name lookup and template declaration and instantiation. The examples are not intended to be programs. Therefore conversion of the examples into compilable programs, in most cases, requires some interpretation. Moreover, a surprising number of examples in the standard contain errors. Some of these errors have been reported with suggested corrections, other errors are still under debate and other erroneous examples seem to have been overlooked. In the next section we describe some further difficulty in converting an example listed in the ISO standard into a test case. In Section 3.2 we discuss the C++ standard core language issues still under investigation by the working committee. In Section 3.3 we describe the *TC1* revisions to the standard.

3.1 Conversion of an example to a possible test case

Figure 3 illustrates an example extracted directly from Clause 14.1 paragraph 3 of the standard. This example is intended to illustrate scope issues about types and variables at global scope as compared to

```

1 class T { /* ... */ }
2 int i;

3 template<class T, T i> void f(T t) {
4     T t1 = i;      // template-parameters T and i
5     ::T t2 = ::i; // global namespace members T and i
6 }

```

Figure 3: *Template example*. This example is from Clause 14 of the ISO C++ standard. However, the example has been assigned *WP* status by the core language working committee.

variables and types at template local scope. Line 1 lists a declaration of class `T` and line 2 lists a declaration of integer `i`. Lines 3 through 6 list a declaration of a template function `f` with two parameters to the template and one parameter to the function. Line 4 in Figure 3 declares an instance, `t1`, of template parameter `T`, initialized to the second template parameter `i`. Line 5 declares an instance of class `T`, declared at the global scope on line 1, passing global variable `i` to the conversion constructor of `T`.

There are several problems in converting the code example in Figure 3 to a test case. The first problem is that the declaration on line 5 uses a conversion constructor in `T` that is not included in the code listing; thus the program, as listed, is ill-formed. The second problem is that the example in Figure 3 is likely to compile on most compilers, even though it is ill-formed, because function `f` is not instantiated. To address these two problems we include a conversion constructor for integers in `T` and we instantiate `f`.

The possible test case corresponding to the example in Figure 3 is illustrated in Figure 4 with the conversion constructor for `T` listed on line 3 and the instantiation of `f` listed on line 13 of `main`. However, the example in Figure 3 and thus the corresponding program in Figure 4 are currently under “*WP*” status, so we do not include this program in our test suite. In the next section we describe the various categories of examples under investigation as C++ Standard Core Language Issues, including an explanation of “*WP*” status.

3.2 The C++ Standard Core Language Issues

The specification for the C++ language was ratified by the ISO standards committee in September of 1998 [3]. The C++ language standard consists of 776 pages describing the core C++ language

```

1 class T {
2 public:
3     T(int n) : number(n) { }
4 private:
5     int number;
6 }
7 int i;

8 template<class T, T i> void f(T t) {
9     T t1 = i;      // template-parameters T and i
10    ::T t2 = ::i; // global namespace members T and i
11 }
12 int main() {
13     f<float, 1.0>(2.5);
14 }

```

Figure 4: *Template test case*. This figure illustrates the modifications to the example illustrated in Figure 3 to convert it into a possible test case.

and the standard C++ library. However, as might be expected for such a large and intricate document, the ISO standard contained issues or examples that require investigation as possible errors. Of these issues, as of this writing there were 411 identified for the core language and 402 identified for the C++ standard library. A constantly evolving discussion of these issues can be found on the newsgroup *comp.std.c++*.

We consider only the 411 core language issues since we are only concerned with conformance of the C++ language and we do not consider conformance of the libraries. These 411 core language issues partition into ten categories and an explanation of each of these ten categories can be found at the URL in reference [4]. We have searched through the 411 language issues and eliminated any test cases that were extracted from examples that fall into these ten categories, with the exception of the category labeled *TC1*. For example, the code listing in Figure 3 is a *WP* issue, a defect report issue that the committee has voted to apply to the current *Working Paper*, or *WP*, which is a draft for a future version of the standard. We do not include any test cases in our study that are extracted from examples in the *WP* issues list.

3.3 Technical Corrigendum 1

The example listed in Figure 5 falls into the *TC1* category of language issues. The *TC1* category describes issues or examples from the standard that are recognized as defects and are included in *Technical*

```

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 5: *A defective example.* This example is taken from Clause 3 of the ISO C++ standard. This example is flawed and has been assigned *TC1* status by the core language working committee.

Corrigendum 1 (TC1), a revision to the standard issued in 2003. Thus, the example listed in Figure 5 is a defect that is officially recognized and ratified by the ISO committee. Unfortunately, we described the example in Figure 5 as a test case that all compilers failed and we received many emails from readers expressing disbelief that any compiler could disambiguate name lookup of *f* listed on lines 1, 3 and 6 of the figure. These readers were quite correct, as verified by the ISO committee.

We do not include any test cases extracted from the C++ standard that fall into any of the ten categories of language issues except for those that fall into the *TC1* category. We do include test cases from the *TC1* category since the committee has ratified these corrections to the standard.

4 The Python test harness

In our previous conformance article, we presented the design and implementation of a Python testing framework that automatically compiled, linked, executed and managed the test execution process. Our framework exploited a Python module, *unittest*, written by Purcell[6], and patterned after the JUnit framework developed by Beck and Gamma[2], and included with Python versions 2.1 and later[7]. We do not repeat the presentation of the framework in this current article.

However, Figure 6 illustrates the constructor for class `CppTestCase`, the class that we use in our framework to wrap test cases. Lines 2 through 27 in the figure illustrate the constructor that initializes a Python array that stores the commands to compile and link programs for each of the eight compilers that we tested. We describe this constructor to expose the command line parameters and flags that

```

1 class CppTestCase(unittest.TestCase):
2     def __init__(self, testfun, fname):
3         unittest.TestCase.__init__(self, testfun)
4         self.compile = [
5             "g++ -Wall -ansi -pedantic-errors -c %s.cpp",
6             "cl /Za /W4 /c %s.cpp",
7             "bcc32 -A -RT -q -w -x -c %s.cpp",
8             "eccp --strict -c %s.cpp",
9             "pgCC -w -Xa -c %s.cpp",
10            "como -c %s.cpp",
11            "icc -ansi -Wall -c %s.cpp",
12            "wcl386 -za -zq -xr -xs -wx -c %s.cpp"
13        ]
14        self.link = [
15            "g++ -o %s.exe %s.o",
16            "cl /nologo /w /Fe %s.exe %s.obj",
17            "bcc32 -q -e %s.exe %s.obj",
18            "eccp --strict -o %s.exe %s.cpp",
19            "pgCC -o %s.exe %s.o",
20            "como -o %s.exe %s.o",
21            "icc -o %s.exe %s.o",
22            "cl /w /Fe %s.exe %s.obj"
23        ]
24        self.fileName = fname
25        self.toPass = not (fname[:4] == "fail")
26        self.hasMain = 0
27        self.directory = os.getcwd()

```

Figure 6: *The Test Case Class.* This figure lists the constructor for the `CppTestCase` class in our Python test harness. This constructor illustrates the command line parameters and flags that we use in testing each of the compilers. The test harness automates the testing process.

we use in testing each of the compilers.

The compilers in our study include *edg 3.2* by Edison Design Group, *Comeau 4.3.2* by Comeau Computing, *Intel 7.1* by Intel Corporation, *PGCC 4.1-2* Workstation C++ compiler by Portland Group Inc., *Visual C++ 7.1* by Microsoft Corporation, *gcc 3.3* by GNU Software Foundation, *Borland 6.0* by Borland Software Corporation³, and *Watcom 1.0* by Open Watcom.

In Figure 6, lines 5 and 15 compile and link programs for *gcc 3.3*, lines 6 and 16 compile and link programs for *VC++ 7.1*, lines 7 and 17 compile and link programs for *Borland 6.0*, lines 8 and 18 compile and link programs for *edg 3.2*, lines 9 and 19 compile and link programs for *PGCC 4.1-2* (the Portland

³The Borland C++ compiler that we tested is version 5.6 of the command line compiler, released with version 6.0 of the C++ Builder IDE.

Compiler	Basic	Conversions	Expressions	Statements	Declarations	Declarators	Classes	Derived	Access	Members	Overloading	Templates	Exceptions	Fails	% Passed
edg 3.2	1	0	0	0	0	0	0	0	0	0	0	1	0	2	99.70
Comeau 4.3.2	1	0	0	0	0	0	0	0	0	0	1	1	0	3	99.55
Intel 7.1	0	0	0	0	0	0	0	0	0	0	2	1	0	3	99.55
PGCC 4.1-2	0	0	0	0	0	0	1	0	0	0	2	3	0	6	99.11
VC++ 7.1	1	0	0	0	2	0	0	0	1	1	2	5	0	12	98.22
gcc 3.3	1	0	0	2	3	5	1	0	6	0	1	6	1	26	96.14
Borland 6.0	0	0	1	0	9	2	1	0	8	3	3	21	1	49	92.73
Watcom 1.0	11	1	4	2	15	4	4	1	9	10	6	76	2	145	78.49
Total Cases	65	2	18	13	76	81	37	33	45	50	54	188	12	674	—

Figure 7: *Conformance of current compilers.* This figure illustrates the results for our conformance tests. The leftmost column lists the compilers reviewed, the topmost row lists the clauses from the ISO C++ standard and the remaining rows list the results of the tests.

Group compiler), lines 10 and 20 compile and link programs for *Comeau 4.3.2*, lines 11 and 21 compile and link programs for *Intel 7.1* and lines 12 and 22 compile and link programs for *Watcom 1.0*.

Lines 24 through 27 of Figure 6 initialize the file name for the program under test, determine whether the test case is positive or negative, initializes a variable that will eventually indicate whether the test case includes a main program and should therefore be linked, and finally sets the directory for the particular clause under test. Further explanation of the Python framework can be obtained from reference [5] and we provide further explanation of the compilers in Section 5 and the compile and link options in Section 5.3.

5 Compiler conformance

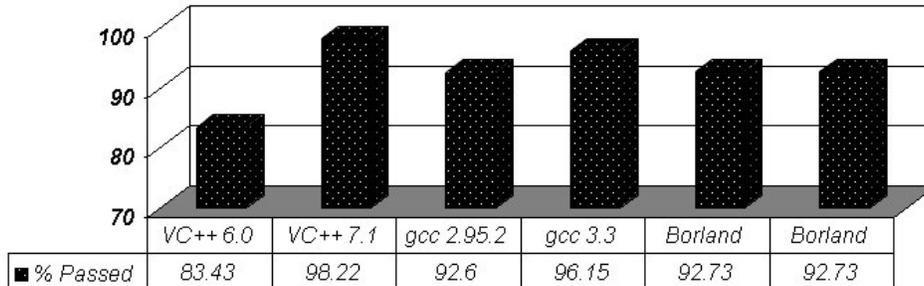
In this section, we apply our Python testing framework to eight popular C++ compilers running on several different platforms. The compilers in our study, described in Section 4, include *edg 3.2*, *Comeau 4.3.2*, *Intel 7.1*, *PGCC 4.1-2*, *Visual C++ 7.1*, *gcc 3.3*, *Borland 6.0*, and *Watcom 1.0*. We chose compilers that target a desktop PC and our goal is to test the C++ language rather than the C++ standard library. We tested the *Visual C++ 7.1*, *Borland 6.0* and *Watcom 1.0* compilers on the Windows XP platforms; all of the rest of the compilers were tested on a system running the Red Hat 9.0 distribution of GNU/Linux.

We have tested the framework using 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 188 test cases for clause 14, containing the largest number of test cases, in 18.2 seconds using the *edg 3.2* compiler and in 7.2 seconds using the *gcc 3.3* compiler on a Dell Precision 530 workstation with a Xeon 1.7 GHz processor and 512 MB of Rambus memory.

In the next section we present our comparison of conformance for each of the eight compilers that we tested. In Section 5.2 we show the progress for three of the compilers that we tested for conformance in our June 2002 article [5]. Finally, in Section 5.3 we show the effects of compiler switches or flags on the relaxation or enhancement of conformance.

5.1 Conformance of current compilers

The table in Figure 7 summarizes our results, where the first column lists the names of the compilers and the columns labeled 3 through 15 list the number of failed test cases for clauses 3 through 15 for each of the respective compilers. The language construct addressed by each clause of the standard is shown at the top of Figure 7, with each construct written vertically at the top of the table. The column labeled *Fails* lists the total number of test cases failed by the respective compiler and the final column, *% Passed*, represents the percentage of test



Compiler	3	4	5	6	7	8	9	10	11	12	13	14	15	Fails	% Passed
VC++ 6.0	7	0	2	3	9	6	3	3	7	4	9	53	6	112	83.43
VC++ 7.1	1	0	0	0	2	0	0	0	1	1	2	5	0	12	98.22
gcc 2.95.2	1	0	1	2	6	6	1	0	9	1	3	15	5	50	92.60
gcc 3.3	1	0	0	2	3	5	1	0	6	0	1	6	1	26	96.15
Borland 5.5	0	0	1	0	9	2	1	0	8	3	3	21	1	49	92.73
Borland 6.0	0	0	1	0	9	2	1	0	8	3	3	21	1	49	92.73
Total Cases	65	2	18	13	76	81	37	33	45	50	54	188	12	674	—

Figure 8: Progress toward conformance of three of the compilers tested in our June 2002 article.

cases that each compiler passed.

The bottom row of the table in Figure 7 lists the number of test cases in each of the respective clauses, with the total number of test cases at 674. For example, column one of the table shows that the *edg 3.2* compiler failed only 1 of the 65 test cases for clause 3, and the *Watcom 1.0* compiler failed 11 of the 65 test cases that we extracted from Clause 3 of the ISO standard.

The final column of Figure 7 shows that the top six compilers passed at least 96% of the test cases including *edg 3.2*, *Comeau 4.3.2*, *Intel 7.1*, *PGCC 4.1-2*, *Visual C++ 7.1*, and *gcc 3.3*. Also, the *Borland 6.0* compiler passed over 92% of the test cases and the open source *Watcom 1.0* compiler passed 78% of the test cases. Considering the intricate examples in the clauses of the standard that exercise complicated and esoteric C++ language constructs, this performance shows that current compiler technology is shaping up well to the monumental task of recognition and compilation of language constructs involving scoping, name lookup, templates and template instantiation, namespaces and exceptions. It is notable that for all compilers except *gcc*, almost half of the test case fails occur for Clause 14 of the standard, which deals with templates.

5.2 Progress toward conformance

In this section, we compare the conformance of three of the compilers that we tested in our previous

article in an effort to measure the progress that these compilers have achieved, since June 2002, toward conforming to the ISO C++ standard. In our previous article, we tested *Visual C++ 6.0*, *gcc 2.95.2* and *Borland 5.5*. In this section, we compare the progress of these three compilers with their current versions: *Visual C++ 7.1*, *gcc 3.3* and *Borland 6.0*.

Figure 8 contains a graph that summarizes the progress and a table that provides detail about the progress of these compilers toward conformance. The results in the columns of the table are similar to those in Figure 7 and each pair of rows compares two versions of each of the three compilers. The final column of the table in Figure 8 shows that *Visual C++ 6.0* only passed 83.43% of the test cases but *Visual C++ 7.1* passed 98.22% of the test cases. This rate of progress toward conformance was not matched by the other two compilers. For example, row three of the table in Figure 8 shows that *gcc 2.95.2* passed 92.60% and row four shows that *gcc 3.3* improved to 96.15%. Rows five and six of the table show that the *Borland 5.5* and *Borland 6.0* compilers passed the same number of test cases.

The bar graph in Figure 8 graphically depicts the progress of these three compilers. The first bar in the graph shows the score for *Visual C++ 6.0*, the lowest score for any test shown in the graph and the second bar shows the score for *Visual C++ 7.1*, the highest score for any test shown in the graph.

Compiler/Flags	Pos	Neg	Fails	% Passed
VC++ 7.1, /Za /W4	5	7	12	98.22
VC++ 7.1, no flags	5	18	23	96.59
gcc 3.3, -Wall -ansi -pedantic-errors	10	16	26	96.15
gcc 3.3, no flags	10	16	26	96.15
edg 3.2, --strict	2	0	2	99.70
edg 3.2, --g++	1	14	15	97.78
Total Cases	407	267	674	—

Figure 9: *The varying effects of compiler flags on three of the compilers tested.*

5.3 Effects of compiler flags on conformance

As a final measure of the compilers that we tested, we now examine the effects of compiler flags on acceptance or rejection of the test cases in our test-suite. Most of the compilers, such as *Visual C++ 7.1*, *gcc 3.3* and *edg 3.2*, include compiler flags or switches that permit the user to relax or enhance the enforcement of conformance to the standard. This relaxation of conformance may then permit programmers to compile legacy code that is not ISO conformant.

Figure 9 reports the results of enforcing or relaxing conformance. The first two rows of the table report the results for *VC++ 7.1* with and without flags respectively, the third and fourth rows report results for *gcc 3.3* with and without flags respectively and the fifth and sixth rows report the results for *edg 3.2* using *--strict* and *--g++* flags respectively. The first column in Figure 9 lists the compiler, the second column lists the number of positive test cases failed, the third column lists the number of negative test cases failed, the fourth column lists the total number of test cases failed and the final column lists the percentage of test cases passed.

The first row shows that the *VC++ 7.1* compiler, using */Za* and */W4* flags, failed 5 positive test cases and 7 negative test cases. The second row shows that, using no flags, the *VC++ 7.1* compiler failed the same number of positive test cases but failed 18 negative test cases. Recall that a negative test case usually fails if it compiles and executes. Thus, the *VC++ 7.1* compiler allows more negative test cases, or non-conforming test cases, to compile and execute using no flags. Rows three and four show that using the *-pedantic-errors* flag with the *gcc 3.3* compiler had no effect on the test cases in our study.

Rows five and six of Figure 9 show that using the *edg 3.2* compiler with the *--g++* flag causes 15

```

1 static void f();
2 static int i = 0;
3 void g() {
4     extern void f(); // internal linkage
5     int i; //2: 'i' has no linkage
6     {
7         extern void f(); // internal linkage
8         extern int i; //3: external linkage
9     }
10 };

```

Figure 10: *Test case from Clause 3.*

negative test cases to fail. The first column of the figure shows that the *VC++ 7.1* and *gcc 3.3* compilers fail the same number of positive test cases independent of the flags used to compile. However, rows five and six show that the *edg 3.2* compiler passes an additional positive test case using the *--g++* flag. This additional test case that *edg 3.2* passes is listed in Figure 10 and the error that causes the test case to fail under strict conformance is a conflict of internal/external linkage of variable *i* listed on lines 2 and 5 of Figure 10. This test case caused trouble for several of the top compilers; some compilers issued an error because the static function *f* on line 4 of the figure does not have a body.

The test case in Figure 11 is the only positive test case that the *edg 3.2* compiler failed under the *--g++* flag. This test case was failed by the four top compilers listed in Figure 7 and may indicate a flaw in *Technical Corrigendum 1* to the standard. Lines 1 through 7 define a template class *A*, with a nested, separate template class *B*; line 8 defines a specialization of *B*. However, there is nothing in the standard that relaxes the access checking on explicit specializations, which would therefore suggest that the example should generate an error [1].

```

1 // Changed wrt TC1 (issue #24)
2 template<class T1> class A {
3     template<class T2> class B {
4     public:
5         void mf();
6     };
7 };
8 template<> template<> class A<int>::B<double>;

```

Figure 11: *Test case from Clause 14.* The top four compilers listed in Figure 7 failed the test case illustrated in this figure, which may indicate a flaw in the *Technical Corrigendum 1* to the standard.

6 Summary

We have presented our approach toward construction of a test suite using examples in the ISO C++ standard and described the work of the committee in addressing standard core language issues. We have used our test suite to provide an approximation of the conformance of eight popular compilers to the C++ standard. We have also evaluated the progress that some of the compilers in our previous evaluation have made toward conformance. Finally, we have illustrated the effects of compiler flags in relaxing the conformance of three popular compilers.

References

- [1] S. Adamczyk. Personal communication, July 2003.
- [2] E. Gamma and K. Beck. Test infected: Programmers love writing tests. Using JUnit to automatically generate test cases, 2001.
- [3] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.
- [4] C++ Standard Core Language Active Issues. <http://anubis.dkuug.dk/jtc1/sc22/wg21/>, July 2003.
- [5] B. A. Malloy, S. A. Linde, E. B. Duffy, and J. F. Power. Testing C++ compilers for ISO language conformance. *Dr. Dobbs Journal*, pages 71–80, June 2002.
- [6] S. Purcell. Python unit testing framework. *documentation freely available at <http://pyunit.sourceforge.net/>*, March 2002.
- [7] Guido van Rossum. *Python Library Reference*. Python Software Foundation, 2001.