Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

C++ Compilers & ISO Conformance

Source Code Accompanies This Article. Download It Now.

- cppiso2.txt
- isocpp_1.zip

Brian, James, and Tanton examine how eight popular C++ compilers measure up to ISO conformance standards.

November 01, 2003
URL:http://www.drdobbs.com/cpp/c-compilers-iso-conformance/184405483

Yes, we are making progress

Brian is an associate professor in the computer science department at Clemson University and can be contacted at malloy@cs.clemson.edu. James is a researcher in the computer science department at the National University of Ireland and can be contacted at jpower@cs.may.ie. Tanton is a software developer for Acxiom Corporation and can be contacted at tanton@deltafarms.com.

Conformance to Standards is becoming recognized as one of the most important assurances compiler vendors can provide to programmers. Conformance enables code portability and wider use of a language and its libraries. However, establishing the conformance of a compiler is difficult—especially for C++, which was slow to develop (with acceptance of a Standard occurring years after the language was introduced).

One approach to 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, constructing such test suites is difficult because there are no central repositories of test cases that conform to the Standard. Of course, you could build test cases using coding examples listed on the Web, but these coding examples are likely designed for specific compilers and, therefore, exhibit bias.

An alternative approach is to extract test cases from examples in the ISO C++ Standard; see ISO/IEC JTC 1. *International Standard: Programming Languages—C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998. Since such examples are usually intended to explain or demonstrate intricate language features—not to be exhaustive—some language features may go untested. Likewise, the examples are not evenly distributed among the language features and some features receive more example focus than others. The template construct, for instance, has attracted special attention in the Standard; thus, a compiler that does not handle a particular template construct may fail a disproportionately higher number of test cases than one that can handle the construct.

But perhaps the greatest difficulty in using the examples in the Standard is that the Standard itself is a work in progress. Since its ratification by the ISO committee in September 1998, 411 core language issues have been raised by the C++ user community. Of these 411 issues, 93 have been addressed by the committee, with these changes incorporated into Technical Corrigendum 1 (TC1), a revision to the Standard issued in 2003. Thus, the ISO C++ Standard is in a state of flux, though clearly moving toward a fixed point.

In this article, we revisit the C++ conformance study we presented in "Testing C++ Compilers for ISO Language Conformance" (*DDJ*, June 2002). In doing so, we provide some measure of conformance to the ISO Standard for eight C++ compilers:

- Borland 6.0 from Borland Software (the Borland C++ compiler we tested is Version 5.6 of the command-line compiler, released with Version 6.0 of the C++ Builder IDE).
- Comeau 4.3.2 from Comeau Computing.
- edg 3.2 from the Edison Design Group.
- gcc 3.3 from the GNU Software Foundation.
- Intel 7.1 from Intel.
- PGCC 4.1-2 Workstation C++ compiler from the Portland Group.

- Visual C++ 7.1 from Microsoft.
- Watcom 1.0 from Open Watcom.

Keep in mind that our results provide an approximate, rather than full, measure of conformance because the examples in the Standard are not exhaustive, are unevenly partitioned across language constructs, and because the Standard is a work in progress.

Extracting Test Cases

We start by describing our approach for extracting test cases from examples in the ISO C++ Standard. We focus only on the core language, as described in Clauses 3-15 of the Standard and do not consider other aspects of the Standard (such as the C++ preprocessor or the Standard C++ Library).

Each clause in the Standard contains C++ examples, most of which include descriptions of the expected behavior or outcomes if the examples are converted to programs. Some of the examples illustrate code that should compile, while others contain code that should not. We attempt to convert the former example into a positive test case and the latter into a negative test case. In other words, if an example contains code that should not compile, the compiler fails the corresponding test case if the test case compiles without a suitable diagnostic. (Some authors refer to positive test cases as "conformance tests" and negative test cases as "deviance tests.")

An example in the Standard can produce many test cases. Some examples expand into multiple positive test cases, while others expand into multiple positive and negative test cases. Listing One is taken from Section 3.4.3 of Clause 3, which specifies rules for qualified name lookup.

Lines 1-4 list class *A* with a static integer *n* and lines 5-9 list function *main*. In *main*, line 6 contains a declaration of the 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; 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; thus, the default lookup rules will find the variable *A* on line 6, rather than the type *A* declared on lines 1-4.

Listing One 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 consists of all of the lines in Listing One except line 8. The negative test case consists of all the lines in Listing One except for line 7. The positive test case passes if it compiles and executes; the negative one passes if it fails to compile or causes the compiler to issue an appropriate 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.

If left unaltered, many of the examples in the Standard 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 nonconforming. 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 100 pages of nonconforming extensions. (All references to gcc here refer to the C++ compiler, not the entire suite of compilers included with gcc.) We have also found variation in nomenclature of include files across vendors.

In some cases, we were able to avoid the problem of nonconforming extensions or variation in include library files if the class or function in the included file was 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 *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 a compiler was not penalized because of nonstandard 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. Listing Two illustrates the minimal stub class *complex* that we used for some of the test cases from Clause 14.

Convergence of the Standard

Our goals in choosing the ISO Standard as the source of our test suite were to:

- Build a test suite that covered the important issues for compilation of C++ programs.
- Obtain test cases that were unbiased toward any particular compiler or vendor.
- Use the Standard as an oracle to determine the validity and outcome of the test cases.

While our extracted test suite meets the first two goals, the third goal is more elusive. The examples described in the Standard are intended to illustrate intricate facets of C++ language syntax or semantics—they aren't 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, others are still under debate, and others seem to have been overlooked.

Listing Three, extracted directly from Clause 14.1 paragraph 3, is intended to illustrate scope issues about types and variables at global scope as compared to variables and types at template local scope. Line 1 lists a declaration of class *T* and line 2 a declaration of integer *i*. Lines 3-6 list a declaration of a template function *f* with two parameters to the template and one parameter to the function. Line 4 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 Listing Three 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 is ill formed. The second problem is that Listing Three 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 instantiate *f*.

The possible test case corresponding to the example in Listing Three is illustrated in Listing Four, with the conversion constructor for *T* listed in line 3, and the instantiation of *f* listed in line 13 of *main*. However, Listing Three (and the corresponding program in Listing Four) are currently under WP (short for "Working Paper," a draft for a future version of the Standard) status, so we do not include this program in our test suite.

Again, the C++ Language Standard consists of 776 pages describing the core C++ language and the C++ Standard Library. However, the ISO Standard contains issues or examples that require investigation as possible errors. At this writing, there are 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 comp.std.c++ newsgroup.) Here, we consider only the 411 core language issues because we are only concerned with conformance of the C++ language, not library conformance. These 411 core-language issues partition into 10 categories (an explanation of each category can be found at http://anubis.dkuug.dk/jtc1/sc22/wg21/). We searched through the 411 language issues and eliminated any test cases extracted from examples that fall into these 10 categories, with the exception of the category labeled "TC1."

Listing Five falls into the TC1 category, describing issues or examples from the Standard that are recognized as defects and are included in TC1. Thus, Listing Five is a defect that is officially recognized and ratified by the ISO committee. Unfortunately, in our June 2002 article, we described Listing Five as a test case that all compilers failed and we received many e-mails from readers expressing disbelief that any compiler could disambiguate name lookup of *f* listed on lines 1, 3, and 6 of the example. These readers were quite correct, as verified by the ISO committee.

The Python Test Harness

In our June 2002 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 unittest, a Python module written by S. Purcell (http://pyunit.sourceforge.net/), 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. An in-depth discussion of the framework can be found in the June 2003 article; the source code for the framework is available electronically (see "Resource Center," page 5).

Listing Six illustrates the constructor for class *CppTestCase*, which we use to wrap test cases. Lines 2-27 illustrate the constructor that initializes a Python array that stores the commands to compile and link programs for each of the compilers we tested. We describe this constructor to expose the command-line parameters and flags that we use in testing each of the compilers.

In Listing Six, 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; 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 24-27 of Listing Six initialize the filename for the program under test, determine whether the test case is positive or negative, initialize a variable that eventually indicates whether the test case includes a main program and should be linked, and set the directory for the particular clause under test.

Compiler Conformance

In applying our Python testing framework to the eight C++ compilers, our goal was to test the C++ language rather than the C++ Standard Library. We tested Visual C++ 7.1, Borland 6.0, and Watcom 1.0 on Windows XP; the other compilers were tested on the Red Hat 9.0 distribution of GNU/Linux.

We tested the framework using Python 1.5 through 2.2. For Python versions prior to 2.1, the unittest module must be downloaded separately. We were able to run 188 test cases for Clause 14 (which contains the most test cases) in 18.2 seconds using the edg 3.2 compiler, and 7.2 seconds using gcc 3.3 on a Dell Precision 530 workstation with a Xeon 1.7-GHz processor and 512 MB of Rambus memory.

Table 1 summarizes our results. The first column lists the names of the compilers. The columns labeled 3-15 list the number of failed test cases for Clauses 3-15 for each of the respective compilers. The language construct addressed by each clause of the Standard is at the top of Table 1, 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 column "%Passed" represents the percentage of test cases that each compiler passed.

The bottom row of Table 1 lists the number of test cases in each of the respective clauses, with the total number of test cases at 674. For example, column 1 shows that the edg 3.2 compiler failed only one of the 65 test cases for Clause 3, while 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 Table 1 shows that the top six compilers passed at least 96 percent 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 percent of the test cases and the Watcom 1.0 compiler passed 78 percent of the test cases. Considering the intricate examples in the clauses of the Standard that exercise complicated and esoteric C++ language constructs, this performance indicates that current compiler technology is shaping up well to the task of recognition and compilation of language constructs involving scoping, name lookup, templates and template instantiation, namespaces, and exceptions. For all compilers except gcc, almost half of the test case failures occur for Clause 14 of the Standard, which deals with templates.

We also compared the conformance of three of the compilers that we tested in our June 2002 article—Visual C++ 6.0, gcc 2.95.2, and Borland 5.5—to measure the progress these compilers have made in conforming to the ISO C++ Standard.

Table 2 summarizes the progress of these compilers toward conformance. The results in the columns of the table are similar to those in Table 1, and each pair of rows compares two versions of each of the three compilers. The final column shows that VC++ 6.0 only passed 83.43 percent of the test cases, but VC++ 7.1 passed 98.22 percent of the test cases. This progress toward conformance was not matched by the other two compilers. For example, row 3 of Table 2 shows that gcc 2.95.2 passed 92.60 percent, and row 4 shows that gcc 3.3 improved to 96.15 percent. Rows 5 and 6 show that the Borland 5.5 and Borland 6.0 compilers passed the same number of test cases.

As a final measure of the compilers, we examined the effects of compiler flags on acceptance/rejection of the test cases in our test suite. Most of the compilers (such as VC++ 7.1, gcc 3.3, and edg 3.2) include compiler flags or switches that let you relax or enhance the enforcement of conformance to the Standard. This relaxation of conformance may then let you compile legacy code that is not ISO conformant.

Table 3 reports the results of enforcing or relaxing conformance. The first two rows of the table report the results for VC++ 7.1 with/without flags, the third and fourth rows report results for gcc 3.3 with/without flags, and the fifth and sixth rows report the results for edg 3.2 using *--strict* and *--g++* flags, respectively. The first column in Table 3 lists the compiler, the second the number of positive test cases failed, the third the number of negative test cases failed, the fourth 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 five positive test cases and seven negative test cases. The second row shows that, using no flags, VC++ 7.1 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 nonconforming test cases, to compile and execute using no flags. Rows 3 and 4 show that using the *-pedantic-errors* flag with the gcc 3.3 compiler had no effect on the test cases in our study.

Rows 5-6 of Table 3 show that using the edg 3.2 compiler with the *--g++* flag causes 15 negative test cases to fail. The first column 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 5-6 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 Listing Seven, 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 Listing Seven. This test case caused trouble for several of the compilers; some issued an error because the static function *f* on line 4 does not have a body.

The test case in Listing Eight 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 in Table 1 and may indicate a flaw in TC1. Lines 1-7 define a template class *A*, with a nested, separate template class *B*; line 8 defines a specialization of *B*. However, as S. Adamczyk of the Edison Design Group indicated, 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.

DDJ

Listing One

```
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 }
```

Back to Article

Listing Two

```
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 } ;
```

Back to Article

Listing Three

```
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 }
```

Back to Article

Listing Four

```
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() f{
13    f <float , 1.0 > (2.5);
14 }
```

Back to Article

Listing Five

```
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 };
```

Back to Article

Listing Six

```
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()
```

Back to Article

Listing Seven

```
1 static void f();
2 static int I=0;
3 void g() {
4   extern void f();  //internal linkage
5   int I; // 2: 'I' has not linkage
6   {
7      extern void f(); // internal linkage
8      extern int I;    // 3: external linkage
9   }
10 }
```

Back to Article

Listing Eight

```
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 <>
9 class A < int > ::B < double >;
```

Back to Article

Compiler	Basic 3	Conversion 4	Expressions 5	Statements 6	Declarations 7	Declarators 8	Classes 9	Derived 10	Access 11	Members 12	Overloading 13	Templates 14	Exceptions 15	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	—

Table 1: Conformance of current compilers.

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	—

Table 2: Progress toward conformance of three of the compilers tested in our June 2002 article.

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	—

Table 3: Varying effects of compiler flags on three of the compilers tested.