A First Look at C++ Program Analyzers

By Scott Meyers and Martin Klaus

Other than minor updates to Table 3, this is a pre-publication version of an article that appeared in the February, 1997, issue of Dr. Dobb's Journal.

C++ has a well-deserved reputation for power and flexibility. It has an equally well-deserved reputation for complexity; its gotchas are legion. For example, omitting a virtual destructor in a base class typically leads to incomplete destruction of derived class objects when they are deleted through base class pointers.

Experienced C++ programmers learn to avoid these kinds of problematic constructs, but experience should not really be necessary: troublesome C++ can often be detected by static analysis, i.e., by tools that parse and analyze C++ source code. Such tools are becoming widely available, and during the summer and fall of 1996, we undertook an investigation to identify these tools and to assess their capabilities. In this article, we summarize the initial results of our investigation.

We were interested in the following questions:

- What tools statically analyze C++ programs and issue warnings about likely trouble spots? By focusing on *static* analysis, we limited our research to tools spiritually akin to `lint`. We explicitly ignored tools designed to detect dynamic (runtime) errors, such as programs that monitor memory usage and report on leaks, etc. Such tools are important, but they offer functionality that complements -- not replaces -- that of static analysis. We also ignored products that focus on lexical issues (e.g., identifier names, indentation style); our interest is in tools that identify constructs that affect program *behavior*.
- How comprehensive are the tools in identifying suspect C++ constructs? C++ has many facets, including data abstraction, inheritance, templates, exception handling, etc., and we wanted to find tools that checked for likely errors in many of these areas. A few tools checked only the C subset of C++, and we ignored those offerings. Our interest is in tools for C++ programmers, and C++ programmers have different needs than C programmers.
- How well do the tools work on real programs? Can they parse real source code? Do they scale well when run on large projects? Are they robust enough to handle complex template instantiations, including those generated by the Standard Template Library (see References)?

This article addresses only the first two questions. Our work on the third question continues as this piece goes to press.

Identifying Products

When we began this project, we were aware of several static analysis tools for C++, but we suspected there were tools we didn't know about. To fill the gaps in our knowledge, we posted a request for information to several Usenix newsgroups, including groups devoted to C++ programming, object-oriented programming, and programming on various platforms. Based on the responses we received, we identified the offerings in Table 1.

Vendor	Product	Supported Under
Abraxas Software	CodeCheck	DOS,Windows,Unix
CenterLine Software	C++Expert	Unix
ConSol Software GmbH	xlint++	Unix
Gimpel Software	Flexe-/PC-Lint	DOS,Windows,Unix
Hewlett Packard	CodeAdvisor	Unix

ParaSoft	CodeWizard	Unix
Productivity through Software	ProLint	Unix
Programming Research Ltd.	QA/C++	Unix
Rational Software	Apex C/C++	Unix

Table 1:Static Analysis Tools for C++

We then attempted to contact the vendors to get information on each product. We were unable to get in touch with ConSol Software GmbH, so we eliminated xlint++ from our study. [Note: after this article went to press, ConSol Software GmbH contacted us, and we have included in Table 3 the information they gave us.] We also eliminated Productivity through Software's ProLint, because it is fundamentally a GUI wrapper around Gimpel Software's (command-line based) FlexeLint, and we had already decided to include FlexeLint in our examination.

That left us with the following seven products:

- CodeCheck from Abraxas Software. CodeCheck is a standalone product that lets programmers use a C-like language to specify what kinds of program analysis should be performed. It comes with several predefined analysis programs, including some for computing program complexity metrics and for identifying unportable code.
- C++Expert from CenterLine Software. Also a standalone product, C++Expert performs static and dynamic analyses of C and C++ programs. Its static checks are drawn from Scott Meyers' *Effective C++* and *More Effective C++* (see References), and its diagnostics contain hypertext links to on-line versions of those books.
- FlexeLint/PC-Lint from Gimpel Software. Another standalone product (the name is FlexeLint for Unix, PC-Lint for DOS, Windows, and OS/2), FlexeLint/PC-Lint is perhaps truest to the classic `lint` tradition. It can check for over 600 potential error conditions in C and C++ source code, including conditions that affect more than one translation unit or that require detailed dataflow analysis.
- CodeAdvisor from Hewlett Packard. CodeAdvisor is a part of HP's SoftBench development environment. It comes able to enforce 23 predefined rules, but users may extend its capabilities by coding new analyses in C++ and linking them in. Source code information is stored in a database, so it is possible to perform checks that involve multiple translation units.
- CodeWizard from ParaSoft. CodeWizard is a standalone product designed to enforce a set of 24 rules selected from *Effective C++*.
- QA/C++ from Programming Research Ltd. Another standalone product, QA/C++ works in two phases. First, it examines C or C++ source code and stores the results in a database. Different Programming Research analysis tools may then be run against the database, and it is these tools that generate warning messages. There is no database API that lets programmers develop their own analyses.
- The Apex C/C++ Development Environment from Rational Software. The Apex environment includes, among other capabilities, the ability to enforce 22 predefined rules for C and C++ programming.

To these choices we added our own noncommercial program, CCEL (see References), purely for purposes of comparison. CCEL began as a research project on static analysis of C++ programs under the direction of one of us (Meyers) and was eventually fully implemented through independent work by the other (Klaus). We added CCEL to our investigation because we were familiar with its capabilities and limitations, and we felt it would be interesting to compare commercial approaches to our research-based initiative.

Approach

Our testing of the tools was broken into three phases:

1. We developed of a set of benchmark rules constraining the structure of C++ programs. For example, one rule is that all base classes must have virtual destructors. We tried to develop a set of rules that was representative of the kinds of rules real programmers would find useful.
2. We contacted the vendor for each tool and asked them which rules their product could enforce. This information proved useful during our later empirical tests, because discrepancies between vendor claims and our empirical findings often identified subtle differences between our rules and those enforced by the vendors.
3. We developed of a set of sample source files seeded with rule violations. We ran each tool on each source file to see whether the seeded rule violation was correctly identified.

Our results yielded a table (Table 3 below) showing how well each tool enforced our benchmark rules on our benchmark programs.

Choosing Rules

One can imagine many ways to compose a set of benchmark rules for C++ programs, but it is difficult to argue that one set is "better" than another. As a result, we made no attempt to develop the "best" set of rules. Instead, we fell back on the fact that one of us (Meyers) has authored two books containing guidelines for C++ programming (*Effective C++* and *More Effective C++*), and we chose nearly all our rules from those books.

This approach is not as gratuitous as it might appear. *Effective C++* and *More Effective C++* have been well-received in the C++ programming community, and one or both form the basis for many sets of corporate coding guidelines. In addition, at least two of the static analysis tools in our investigation are based on these books. Finally, by drawing our rules from well-known and easily accessible sources, we avoided the need to explicitly justify individual rules in our benchmark set. Instead, the justification for nearly every rule is available in the books, and we simply refer to the appropriate book location as the rationale for each rule.

We chose 36 rules (34 of which were drawn from *Effective C++* or *More Effective C++*), which we divided into eight categories. These rules are listed in Table 2.

General

1	E 1	Use const instead of #define for constants at global and file scope.
2	M 2	Use new-style casts instead of C-style casts.
3	M 3	Don't treat a pointer to Derived[] as a pointer to Base[].

Use of new and delete

4	E 5	Use the same form for calls to new and delete. (In general, this calls for dynamic analysis, but static analysis can catch some special cases, e.g., calls to new in ctors and to delete in dtors.)
5	E 6	When the result of a new expression in a ctor is stored in a dumb pointer class member, make sure delete is called on that member in the dtor.
6	E 9	Avoid hiding the default signature for operator new and operator delete.

Constructors/Destructors/Assignment

7a	E11	Declare a copy constructor for each class declaring a pointer data member.
7b	E11	Declare an assignment operator for each class declaring a pointer data member.
8	E12	Initialize each class data member via the member initialization list.
9	E13	List members in a member initialization list in an order consistent with the order in which they are actually initialized.
10	E14	Make destructors virtual in base classes.
11	E15	Have the definition of operator= return a reference to *this. (Note: this says nothing about declarations.)
12a	E16	Assign to every local data member inside operator=.
12b	E16	Call a base class operator= from a derived class operator=.
12c		Use the member initialization list to ensure that a base class copy ctor is called from a derived class copy ctor.
13		Don't call virtual functions in constructors or destructors.

Design

14	E19	Use non-member functions for binary operations like +-/* when a class has a converting ctor.
15	E20	Avoid public data members.
16	E22	Use pass-by-ref-to-const instead of pass-by-value where both are valid and the former is likely to be more efficient.
17a	E23	Have operators like +-/* return an object, not a reference.
17b	M 6	And make those return values const.
18	E25	Don't overload on a pointer and an int.

19	M33	Make non-leaf classes abstract.
20	M24	Avoid gratuitious use of virtual inheritance, i.e., make sure there are at least two inheritance paths to each virtual base class.

Implementation

21	E29 E30	Don't return pointers/references to internal data structures unless they are pointers/references-to-const.
22	M26	Never define a static variable inside a non-member inline function unless the function is declared extern. [Footnote: In July 1996, changes to the nascent standard for ANSI/ISO C++ obviated the need for this rule, at least on paper. However, the need still exists in practice, because many compilers continue to heed the older rules that can lead to duplicated variables in inline non-member functions.]
23		Avoid use of "..." in function parameter lists.

Inheritance

24	E37	Don't redefine an inherited nonvirtual function.
25	E38	Don't redefine an inherited default parameter value.

Operators

26	M 5	Avoid use of user-defined conversion operators (i.e., non-explicit single-argument ctors and implicit type conversion operators).
27	M 7	Don't overload &&, ||, or ,.
28	M 6	Make sure operators ++ and -- have the correct return type.
29	M 6	Use prefix ++ and -- when the result of the increment or decrement expression is unused.
30	M22	Declare op= if you declare binary op (e.g., declare += if you declare +, declare -= if you declare -, etc.). One way to satisfy this constraint is by providing a template that yields the appropriate function.

Exceptions

31	M11	Prevent exceptions from leaving destructors.
32	M13	Catch exceptions by reference.

Table 2: Benchmark Rules

Each rule begins with its rule number, followed by a reference to the Item number in either *Effective C++* (E) or *More Effective C++* (M) from which it is derived. The text of the rule is often different from the text of the book Item, because the book Items tend to be worded too generally to be checked.

Some of the rules may seem controversial, especially in light of the C++ found in many popular class libraries. Rule 15 (no public data members), for example, is widely violated in the Microsoft Foundation Classes, while almost no library adheres to Rule 19 (make all non-leaf classes abstract). With the exceptions of Rules 13 and 23 (which, in this article, we hope are self-explanatory), *Effective C++* and *More Effective C++* offer firm technical foundations for each rule. We believe it is therefore important that programmers be able to enforce those constraints, even if the majority of programmers choose not to. Furthermore, our decision to include rules that are commonly violated helps us evaluate the effectiveness of the tools' filtering capabilities. (We do not report on this aspect of the tools in this paper, but it is an important consideration in the practical application of any tool; see below.)

Benchmark Programs

For each of our 36 rules, we developed a trivial source file seeded with a violation of the rule. We then executed each tool on each source file to see if the tools correctly identified the seeded errors. These source files were truly *trivial* -- many were under 10 lines long.

Our goal with these files was not to provide a realistic test of the tools, it was just to see whether the tools could identify rule violations in the simplest of cases. (Sometimes this backfired and yielded misleading results -- see below.)

As an example, here is the complete source code for the file used to test rule 20:

```
//    20  M24  Avoid gratuitious use of virtual inheritance, i.e., make
//             sure there are at least two inheritance paths to each
//             virtual base class.

class Base { int x; };
class Derived: virtual public Base {};

Derived d;
```

Compilers Versus Special Tools

Several people responded to our request for information on static analysis tools by remarking that they found little need for such tools. Instead, they wrote, they relied on their compilers to flag conditions that were likely to lead to trouble. The following comment was typical:

I find GNU G++ with -ansi -pedantic -Wall -O flags useful.

In fact, the Gnu compiler was singled out by more than one writer as being especially good at warning about troublesome C++. This piqued our curiosity about compiler warnings. How many of our candidate rules would compilers identify?

To find out, we submitted our benchmark programs to five compilers, in each case enabling as many warnings as possible. As Table 3 shows, the results were disappointing. Even g++ identified at most two of the 36 rule violations, and three of the compilers identified none. This confirmed our impression (based on our experience as C++ programmers) that compilers are good at many things, but identifying legal, but potentially troublesome, C++ source code is not one of them.

Specifying Constraints

The tools in our study allow programmers to specify what conditions to check for in one of two ways. Most tools follow the `lint` model, whereby the tool is created with the ability to enforce some set of predefined constraints, and programmers turn these constraints on or off. There is no way to extend the capabilities of such tools. For example, a tool is either capable of detecting that an exception may leave a destructor (Rule 31) or it's not. If it's not, there is no way for a tool user to add that capability.

A different approach, one employed by Abraxas' CodeCheck, HP's CodeAdvisor, and our research program CCEL, is to provide tool users with a language in which to express constraints of their own. Such tools may or not be useful "out of the box" (it depends on the existence and utility of predefined rule libraries), but they can be extended to check for new, user-defined conditions. This approach is more powerful, but, as in the case of C++ itself, complexity often accompanies power. The power is inaccessible until one has mastered the constraint expression language. Furthermore, the addition of user-defined constraints may affect an analysis tool's performance, because enforcement of such constraints may require arbitrary amounts of time, memory, or other resources.

We made no attempt to master the various constraint expression languages used by the different tools, but the examples we saw (see the appendix) reinforced the lessons we learned during the design and implementation of CCEL: it's hard to design a language for expressing constraints on a language as feature-filled as C++, and given such a constraint language, it's nontrivial to learn to use it. Abraxas, for example, reports that it takes between 3 and 6 months to become proficient in the CodeCheck constraint language. Most of their customers hire specialists to compose their rules instead of learning to write the rules themselves.

Most programmable tools attempt to offer the best of both worlds by shipping with a set of predefined rule libraries that check for commonly-desired constraints. This eliminates the need to write rules to cover common constraints.

Results And Discussion

The results of running the various tools on the collection of benchmark programs is shown in Table 3.

	Compilers					**Static Analysis Tools**								
Product	**Borland C++**	**GNU g++**	**Visual C++**	**SunSoft C++**	**Symantec C++**	**CodeWizard**	**PC-Lint**	**CodeCheck**	**QA/C++**	**CCEL**	**CodeAdvisor**	**C++Expert**	**Apex C++**	**xlint++**
Version	5.0	2.7.2	10.00.5270	SC3.0.1	7.20B1	1.0	7.00c	6.04	3.0.1	N/A	C.05.00	1.0	2.0.6C	Not Tested
Options	-w	-ansi -Wall	-W4	+w2	-w-	unsuppress all	-w4	Note 14	(Defaults)	N/A	(Defaults)	(Defaults)	(Defaults)	Not Tested
Tested Under	NT 3.51	SunOS 4.1.4	NT 3.51	Solaris 2.5	NT 3.51	Solaris 2.5	NT 3.51	NT 3.51	Solaris 2.5	Solaris 2.5	Solaris 2.5	Solaris 2.5	Solaris 2.5	Not Tested
Rule 1	-	-	-	-	-	-	RC	X	X	X	CWP	-	-	C
Rule 2	-	-	-	-	-	-	-	CWP	X	-	-	-	-	-
Rule 3	-	-	-	-	-	-	Note 5	CWP	-	-	CWP	CD	-	-
Rule 4	-	-	-	-	-	Note 4	RC	CWP	Note 4	-	CWP	CD	-	-

Rule 5	-	-	-	-	-	X	X	CWP	-	-	CWP	CD	-	-
Rule 6	-	-	-	-	-	X	RC	CWP	-	X	CWP	X	Note 3	-
Rule 7a	-	-	-	-	-	X	Note 6	CWP	X	X	Note 6	X	-	C
Rule 7b	-	-	-	-	-	X	Note 6	CWP	X	X	Note 6	X	-	C
Rule 8	-	-	-	-	-	Note 3	X	Note 11	X	-	CWP	Note 13	-	C
Rule 9	-	X	-	-	-	X	X	Note 11	X	-	Note 7	X	-	C
Rule 10	-	Note 2	-	-	-	Note 3	Note 10	Note 2	Note 2	X	Note 2	Note 3	X	C
Rule 11	-	-	-	-	-	Note 3	RC	CWP	Note 3	-	CWP	X	-	-
Rule 12a	-	-	-	-	-	X	RC	CWP	-	-	CWP	-	-	C
Rule 12b	-	-	-	-	-	-	RC	CWP	-	-	CWP	-	-	-
Rule 12c	-	-	-	-	-	-	Note 7	Note 11	X	-	CWP	-	-	-
Rule 13	-	-	-	-	-	-	X	CWP	-	-	X	-	-	-
Rule 14	-	-	-	-	-	-	RC	CWP	-	X	CWP	-	-	-
Rule 15	-	-	-	-	-	X	-	CWP	X	X	CWP	X	X	C
Rule 16	-	-	-	-	-	X	RC	CWP	Note 3	X	CWP	Note 3	-	-
Rule 17a	-	-	-	-	-	-	RC	CWP	-	X	CWP	-	-	-
Rule 17b	-	-	-	-	-	-	-	CWP	-	X	CWP	-	-	-
Rule 18	-	-	-	-	-	Note 3	-	CWP	-	X	CWP	X	-	C
Rule 19	-	-	-	-	-	-	-	CWP	-	X	CWP	-	-	-
Rule 20	-	-	-	-	-	-	RC	-	-	X	CWP	-	-	-
Rule 21	-	-	-	-	-	X	Note 3	CWP	Note 3	-	CWP	-	-	-
Rule 22	Note 1	-	-	-	-	-	-	CWP	Note 7	-	CWP	-	-	-
Rule 23	-	-	-	-	-	-	-	X	X	X	CWP	-	-	C
Rule 24	-	-	-	-	-	X	X	CWP	-	X	Note 3	X	-	C
Rule 25	-	-	-	-	-	X	RC	-	-	X	CWP	X	-	-
Rule 26	-	-	-	-	-	-	-	CWP	Note 3	-	Note 12	-	-	-
Rule 27	-	-	-	-	-	-	RC	CWP	-	X	CWP	X	-	C
Rule 28	-	-	-	-	-	-	RC	CWP	-	X	CWP	-	-	-
Rule 29	-	-	-	-	-	-	RC	CWP	X	-	-	-	-	-
Rule 30	-	-	-	-	-	-	-	CWP	-	X	CWP	-	-	C
Rule 31	-	-	-	-	-	-	-	CWP	-	-	-	-	-	-
Rule 32	-	-	-	-	-	-	RC	CWP	-	-	-	X	-	-

Table 3: Results of Submitting Sample Programs to Tools

Legend

-	The tool failed to detect violations of this rule on our benchmark programs.
X	The tool detected the seeded rule violation in our benchmark programs.
C	The vendor claims the tool will detect violations of this rule, but we were unable to acquire a copy of the tool to verify the claim.
CD	The vendor claims the tool will detect violations of this rule *dynamically*, but we were unable to acquire a copy of the tool to verify the claim.
CWP	The vendor claims that users can program the tool to detect violations of the constraint. We did not attempt to verify the claim.
RC	One or more readers claim that versions of the tool released after the article was published will detect violations of this rule. We did not attempt to verify the claim.
Note 1	Warning issued that functions containing static variables are not expanded inline.
Note 2	Will note a nonvirtual destructor in classes with virtual functions.

Note 3	Vendor claims the condition will be detected, but tests exposed at least one failure to do so.
Note 4	A diagnostic is issued only when the calls to new and delete are within the same function.
Note 5	Erroneously diagnosed an error in a valid source file.
Note 6	Doesn't diagnose this particular condition, but uses similar heuristics to identify classes where assignment operators and copy constructors should be declared.
Note 7	A diagnostic is issued that hints at the problem, but the problem is not directly identified.
Note 8	A diagnostic is issued only when the member is initialized in the body of the constructor.
Note 9	(This note is not used.)
Note 10	Attempts to warn about this only when the condition would cause a runtime problem, but tests showed at least one failure to do so.
Note 11	The vendor claims that users can program the tool to detect violations of the constraint, but technical support was unable to describe how when asked.
Note 12	A constraint to enforce the rule can be written, but the HP parser doesn't yet support `explicit`.
Note 13	This condition isn't detected, but use of uninitialized memory is dynamically detected.
Note 14	"-Rcplus" used, but Abraxas describes this as a "tutorial and example file" not designed for production use.

Several features in the data are of interest. First, no tool was able to enforce all our 36 benchmark rules, not even the tools supporting user-defined constraints. Thus, even the best of currently available tools offers only partial coverage of C++. This is especially noteworthy, because our benchmark rules themselves failed to exercise all major language features; templates are a particularly obvious omission.

Second, the number of benchmark rules that can be enforced without programming -- i.e., "out of the box" -- is at most 17 of 36. (CCEL supports 19, but CCEL is a research project, not a commercial product). If we speculate that our set of benchmark rules is somehow representative of the kinds of constraints real programmers might want to enforce, this suggests that current tools cover at best only about half those constraints. Of course, automatic enforcement of half a set of requirements is better than no enforcement at all, but the data in Table 3 suggest that there is much room for increased language coverage by static analysis tools for C++.

Third, it is not uncommon for there to be subtle mismatches between a benchmark rule and the conditions detected by the analysis tools. In most cases, this is an outgrowth of the vendors' attempts to avoid generating warning messages when no truly harmful condition exists. For example, consider again Rule 10:

Make destructors virtual in base classes.

Many programmers consider this rule too aggressive, and a common alternative form of the same rule is this:

Make destructors virtual in classes containing virtual functions.

This form has the advantage that no virtual table pointer is added to a class simply to satisfy the rule. (This is the rule variant that's employed by the Gnu C++ compiler, by HP's CodeAdvisor, and by Programming Research's QA/C++.)

The motivation for this rule (in any form) is that the following code is generally harmful if the base class lacks a virtual destructor:

```
class B { ... };             // base class;  assume no virtual dtor
class D: public B { ... };   // derived class
```

```
void f(B *p);               // f is some function taking a B*

D *pd = new D;              // pd points to a D

f(pd);                      // pass pd to f, binding pd to p in f

void f(B *p)
{
  delete p;                 // this calls only B's dtor, not D's!
}
```

In truth, this code is only harmful if one or more of the following conditions holds:

- D has a destructor
- D has data members that have destructors
- D has data members that contain data members (that contain data members...) with destructors

At least one tool vendor attempts to issue a diagnostic only if these more stringent conditions exist, and the conditions do not exist in our test program:

```
// test program for rule 10

class Base {};
class Derived: public Base {};

int main()
{
  Base *pb = new Derived;
  delete pb;

  return 0;
}
```

The tool in question thus issues no diagnostic on our sample program, but if class `Derived` were nontrivial, the tool might issue a warning.

This more precise analysis should be beneficial for users, because a diagnostic should be issued only if a problem truly exists. However, the rules of C++ can be both complicated and unintuitive, and their subtlety can cut both ways. In the case of the vendor attempting to check for the more detailed conditions outlined above, the test for data members with destructors in the derived class was omitted. Hence, though the tool avoids issuing warnings in harmless cases, it also avoids issuing warnings in some rare, but harmful cases. Yet these are precisely the cases in which static analysis tools that correctly understand the detailed rules of C++ are most useful!

Another tool had trouble issuing correct diagnostics when compiler-generated functions -- default constructors, copy constructors, assignment operators, and destructors (especially derived class destructors) -- were involved. Because of the minimalist nature of our test cases, our programs had many instances of such functions, and this led to incorrect results from some tools.

Whether such shortcomings would cause problems when the tools are applied to real programs is unknown, but it hints at a deeper problem we found with some of the tools: the vendors didn't seem to understand the subtleties of C++ as well as they need to. We believe that vendors of C++ analysis tools must understand C++ as well as compiler vendors, but based on our experience with the tools in this study, we must report that such expertise cannot yet be taken for granted.

Caveats

The results in Table 3 provide insight into the state of existing `lint`-like tools for C++, but it is important to recognize what they do not show. We were interested only in the capability of such tools to handle the "++" part of C++, but most of the tools also provide significant other capabilities. For example:

- Most tools also check the "C" part of C++ -- some of them quite extensively. This can be very useful. By limiting our tests to the C++-specific capabilities of the tools, we were able to sharpen our focus, but we also screened out the majority of some tools' functionality.
- Many tools offer stylistic and lexical checks in addition to the semantic issues we looked at. For example, if you wish to ensure that classes never use the default access level of `private` but instead declare it explicitly, at least one tool will note violations of that constraint.
- Some tools offer complementary analyses in addition to checking coding "style". For example, Programming Research's QA/C++ can calculate various program complexity metrics.

In addition, our set of benchmark rules was far from exhaustive. Some vendors check for C++-specific conditions we didn't consider, and Table 3 says nothing about such capabilities.

All this is to say that Table 3 is anything but a "Buyer's Guide." Furthermore, there are many non-technical characteristics of analysis tools you should consider before deciding which, if any, is suitable for your circumstances. The following questions immediately come to mind:

- How easy is it to install, configure, and use the tool? These factors are especially important for tools with equivalent or close to equivalent capabilities. For example, Gimpel Software's FlexeLint and Productivity Through Software's ProLint use the same underlying analyis engine, but offer quite different user interfaces.
- How easy is it to filter out unwanted diagnostics? The traditional Achilles Heel of `lint`-like tools is an unacceptable signal to noise ratio, so it's important that users be given fine-grained control over what code is analyzed and which diagnostics appear. In fact, some vendors deliberately decided to avoid offering checks for some conditions (e.g., the use of preprocessor macros to define constants -- our Rule 1), because they felt it would be more bothersome than useful to their customers. (Respondants to our newsgroup postings indicated that a bad signal to noise ratio is a continuing problem, even with some of the tools considered here.)
- How robust and up-to-date is the C++ parser? To be maximally useful, a C++ analyzer must parse exactly the same language as the compiler(s) you use. It's particularly frustrating if the analyzer rejects code your compiler accepts.
- Can the tool handle large projects, e.g., those with multiple libraries, with hundreds or thousands of source files, etc? Based on the responses we got from our Usenet postings, the answer is too often that it cannot.
- Is the documentation complete, accurate, accessible, and comprehensible?
- Does the vendor offer adequate customer service, including technical support?
- How well-established is the vendor? Is the vendor likely to continue to support the tool for years to come?

Our study considered none of these issues.

Finally, it is important to bear in mind that the results in Table 3 are based on tests we performed in August and September, 1996. Virtually all of the tools we examined are under active development, so it's likely that new versions of the tools exist even as you read this report. For example, we know that Abraxas is currently beta-testing a set of predefined constraints derived from material in Meyers' books, and we know that CenterLine and Rational are planning upgrades to C++Expert and Apex, respectively, that will allow users to define new constraints. Other tool vendors are similarly active. The data in Table 3 represents a mere snapshot of the commercial state of the art in September, 1996, and that snapshot is, alas, somewhat out of date by the time you see it.

Summary

A number of analysis tools is now available that read C++ source code and warn about possible behavioral problems. They cover varying aspects of C++, though none offers truly comprehensive coverage of the language. Based on simple tests, we believe that many dangerous C++ constructs can be detected, though the complexity of C++ leads to incorrect behavior on the part of some tools, especially where compiler-generated functions are concerned. C++ analysis tools are under active development, and it is likely that the data in this article fails to accurately reflect the current capabilities of the tools we examined. If you are interested in static analysis tools for C++, we encourage you to contact the vendors in Table 1, conduct your own tests, come to your own conclusions, and share them with us.

Acknowledgements

We are grateful to Jill Huchital and to an anonymous reviewer for their comments on drafts of this article, and we thank the editorial staff at Dr. Dobb's Journal for helping us publish our findings much more quickly than is usually possible.

References

Scott Meyers, *Effective C++*, Addison-Wesley, 1992.

Scott Meyers, *More Effective C++*, Addison-Wesley, 1996.

Scott Meyers, Carolyn K. Duby, and Steven P. Reiss, "Constraining the Structure and Style of Object-Oriented Programs," *Principles and Practice of Constraint Programming*, MIT Press, 1995. An earlier version of this paper is available from the Brown University Technical Report server.

David R. Musser, and Atul Saini, *STL Tutorial And Reference Guide*, Addison-Wesley, 1996.

Appendix: Constraint Expression Languages

As an example of the different ways in which constraints on C++ programs may be expressed, consider our Rule 10. In English, the constraint is: *Make destructors virtual in base classes*. When this constraint is formalized and made amenable to enforcement by computer, it quickly becomes more complicated.

CCEL offers a fairly succinct way to express this constraint, because CCEL was specifically designed to make the expression of constraints like this straightforward. Still, CCEL's formal nature makes it wordier than English. (It also makes it more precise.) This is Rule 10 in CCEL:

```
BaseClassDtor (
 Class B;
 Class D | D.is_descendant(B);
```

```
    Assert(MemberFunction B::m; | m.name() == "~" +  B.name() && m.is_virtual());
   );
```

CCEL embodies a *declarative* approach to the specification of constraints: you specify *what* you want to enforce, not *how* to enforce it. This CCEL constraint can be read like this:

For all classes B and all classes D that inherit from B, B must have a member function m such that m is a destructor and m is virtual.

Compared to CCEL, all the programmable analysis tools we investigated are long-winded. That's because they employ a *procedural* approach to constraint specification: you write code specifying *how* to go about detecting violations of the constraints you define. In practice, a procedural approach is more powerful, but it's also more complicated.

For example, Abraxas, which offers a C-like programming language for new constraints, suggests the following implementation for our Rule 10:

```
int isBaseClass;
int inBase;
int inClassHead;
char tagList[ MAXLIST ];

if ( dcl_virtual )
{
    if ( dcl_base == DESTRUCTOR_TYPE )
    {
        if ( tagList[ 0 ] == 0 )
        {
            i = 0;
            while ( i <= strlen( tag_name() ) )
            {
                tagList[ i ] = tag_name()[ i ];
                i++;
            }
        }
        else
        {
            p = tagList;
            while ( 1 )
            {
                if ( p[ 0 ] == 0 )
                {
                    i = 0;
                    while ( i <= strlen( tag_name() ) )
                    {
                        p[ i ] = tag_name()[ i ];
                        i++;
                    }
                    break;
                }
                else
                {
                    p = p + strlen( p ) +1;
                }
            }
        }
    }
}

if ( keyword( "class" ) )
{
    if ( strcmp( prev_token(), "<" ) != 0 )
    {
        inClassHead = TRUE;
    }
}

if ( op_colon_1 )
{
    if ( inClassHead )
    {
        inBase = TRUE;
    }
}
```

```
if ( op_separator )
{
    if ( inBase )
    {
        if ( strcmp( prev_token(), ">" ) != 0 )
        {
            isBaseClass = TRUE;
        }
    }
}

if ( op_open_angle )
{
    if ( inBase )
    {
        if ( angleLevel == 0 )
        {
            isBaseClass = TRUE;
        }
        angleLevel++;
    }
}

if ( op_close_angle )
{
    if ( inBase )
    {
        angleLevel--;
        if ( angleLevel == 0 )
        {
            isBaseClass = FALSE;
        }
    }
}

if ( isBaseClass )
{
    p = tagList;
    while ( p[ 0 ] != 0 )
    {
        if (strcmp( p, prev_token() ) == 0 )
        {
            break;
        }
        else
        {
            p = p + strlen( p ) + 1;
        };        }
    }
    if ( p[ 0 ] == 0 )
    {
        warn( 1401, "The destructor of class %s should be declared as virtual.", tag_name() );
    }
}

if ( tag_begin )
{
    inBase = FALSE;
    inClassHead = FALSE;
}
```

HP's CodeAdvisor uses C++ as an extension language. New constraints are implemented via classes inheriting from the predefined `Rule` class. These classes are then compiled and linked to a runtime library. Here is our implementation of Rule 10:

```
class Rule_VirtDtorInBaseClass : public Rule
{
public:
      virtual int kindMask() const;
      virtual Language langMask() const;
      void check(SymbolTable *, const Symbol &);
      virtual const char *errorMess() const;
      virtual const char *name() const;
};
```

```
// This rule is invoked for all "Tag" objects.  Tags include all
// compound objects, such as Classes, Templates, Structs, Unions, and Enums.
// Can't specify a kindMask() of KIND_CLASS, since check() is invoked
// only on Symbol objects.  Class is not a Symbol; Tag is.

int Rule_VirtDtorInBaseClass::kindMask() const
{ return 1 << KIND_TAG; }

// Returns a mask of the language(s) this rule applies to.

Language Rule_VirtDtorInBaseClass::langMask() const
{ return LANGUAGE_CPP; }


// Find all base classes not declaring a virtual destructor.

void Rule_VirtDtorInBaseClass::check( SymbolTable *, const Symbol &sym)
{
      Tag tag;
      Class cl;

      // Don't want to check instances. The !tag.ClassType(cl) call also rejects enums.

      if (!sym.SymbolToTag(tag) || !tag.ClassType(cl) || IS_INSTANTIATED(cl.Attrib()))
              return;

      // Reject structs and unions, which are also represented as Classes.

      if (WAS_STRUCT(cl.Attrib()) || WAS_UNION(cl.Attrib()))
              return;

      // Get possible base classes and derived classes.

      ATTRIBUTE_ITERATOR(Tag) bcl=cl.BaseClasses();
      ATTRIBUTE_ITERATOR(Tag) dcl=cl.DerivedClasses();

      // Reject derived classes.

      ITERATE_BEGIN(bcl) { return; } ITERATE_END(bcl);

      // Check only base classes.

      ITERATE_BEGIN(dcl) {

              FunctionMember fm;

              char dtor_name[256];
              char msg[1024];

              // Destructors can be identified by their name.

              strcpy(dtor_name, tag.Name());
              strcat(dtor_name, "::~");
              strcat(dtor_name, tag.Name());
              strcat(dtor_name, "()");

              // Base class must declare a destructor member function that is virtual.

              if (cl.FindFunctionMember(dtor_name, fm) && IS_VIRTUAL(fm.Attrib())) {
                      return;
              }

              // Issue the violation message.

              sprintf(msg, "Baseclass '%s' should declare a virtual destructor", tag.Name());
              violation(tag, msg);
              return;

      } ITERATE_END(dcl);
}

// Returns a one-line summary of the violation with no instance-specific information.

const char *Rule_VirtDtorInBaseClass::errorMess() const
```

```
{
      return("Base class must declare a virtual destructor.");
}

// Returns the name of this rule.

const char *Rule_VirtDtorInBaseClass::name() const
{
      return("Rule_VirtDtorInBaseClass");
}

static Rule_VirtDtorInBaseClass instance;
```

Finally, here's Rule 10 in the constraint expression language CenterLine plans to release in an upcoming version of C++Expert. This code was provided by CenterLine:

```
advBool advSMNonvirtualDestructor::DoesEntityViolateRule
(const advClassStructOrUnionType& t,
 advDiagnostic& d) const
{
  advBool result = adv_false;

  do {
    if (t.HasBaseClasses() && t.HasDestructor()) {
      if (t.Destructor()->IsCompilerGenerated()) {

      d << "Class type:" << t <<
      "Base classes without virtual destructors:";

      //  Make sure that all direct
      // bases have virtual destructors.  (There's no need to check
      // indirect bases since they will have been checked previously.)
      class CheckForBaseClassesWithNonvirtualDestructors :
      public
      advForEachFunction<advClassStructOrUnionType::BaseClassInfo>        {
      public:
      CheckForBaseClassesWithNonvirtualDestructors(advDiagnostic& d,
                                                   advBool& result)
        : d_(d), result_(result) {}

      virtual void operator()(const
                              advClassStructOrUnionType::BaseClassInfo& b,
                              advBool&) {
        if (!b.base_class_.HasDestructor() ||
            !b.base_class_.Destructor()->IsVirtual()) {
          // A base class with a non-virtual destructor!
          result_ = adv_true;

            // Add the offending base class to the diagnostic so that
            // the user will know what was wrong.
          d_ << b.base_class_;
        } // if
      }

      private:
      advDiagnostic& d_;
      advBool& result_;
      } check_for_base_classes_with_nonvirtual_destructors(d, result);

      (void)
      t.ForEachBaseClass
      (check_for_base_classes_with_nonvirtual_destructors);
    } // if
  } while (adv_false);

  return result;
}
```