Constraining the Structure and Style of Object-Oriented Programs

Scott Meyers[1]
Carolyn K. Duby[2]
Steven P. Reiss[3]

Technical Report No. CS-93-12
April 1993

[1]Brown University Box 1910, Providence, RI 02912
[2]Cadre Technologies, Inc. 222 Richmond Street,Providence,RI 02903
[3]Brown University Box 1910, Provodence, RI 02912

Constraining the Structure and Style of Object-Oriented Programs*

Scott Meyers
Dept. of Computer Science
Brown University, Box 1910
Providence, RI 02912
sdm@cs.brown.edu

Carolyn K. Duby
Cadre Technologies, Inc.
222 Richmond Street
Providence, RI 02903
ckd@cadre.com

Steven P. Reiss
Dept. of Computer Science
Brown University, Box 1910
Providence, RI 02912
spr@cs.brown.edu

Abstract

Object-oriented languages fail to provide software developers with a way to say many of the things about their systems that they need to be able to say. To address this need, we have designed and implemented a language for use with C++ that allows software developers to express a wide variety of constraints on the designs and implementations of the systems they build. Our language is specifically designed for use with C++, but the issues it addresses are applicable to other object-oriented languages, and the fundamental software architecture used to implement our system could be applied without modification to similar constraint languages for other object-oriented programming languages.

1 Introduction

C++ is an expressive language, but it does not allow software developers to say many of the things about their systems that they need to be able to say. In particular, C++ offers no way to express many important constraints on a system's design, implementation, and stylistic conventions. Consider the following sample constraints, none of which can be expressed in C++:

- *The member function M in class C must be redefined in all classes derived from C.* This is an example of a **design constraint**, because the constraint is specific to a particular class, C, and a particular function in that class, M. This kind of constraint is common in general-purpose class libraries. For example, the NIH class library [6] contains many functions which must always be redefined if the library is to function correctly.

- *If a class declares a pointer member, it must also declare an assignment operator and a copy constructor.* This is an example of design-independent **implementation constraint**. Failure to adhere to this constraint almost always leads to incorrect program behavior [16].

- *All class names must begin with an upper case letter.* This is an example of one of the most common kinds of **stylistic constraint**. Most software development teams adopt some set of naming conventions that developers are required to follow.

Constraints such as these exist (usually only implicitly) in virtually every system implemented in C++, but different systems require very different sets of constraints. That fact makes it untenable for C++ compilers to search for constraint violations. Our approach to this problem is the development of a new language, CCEL ("Cecil") — the C++ Constraint Expression Language — a language for use with C++ that allows software developers to specify a wide variety of constraints and to automatically detect violations of those constraints.

We took as our original inspiration the lint tool for C programmers, but we quickly discovered that the kinds of errors C programmers need to detect are qualitatively different from the errors that C++ programmers

*This paper was originally presented at PPCP93 — the First Workshop on Principles and Practice of Constraint Programming, April 1993.

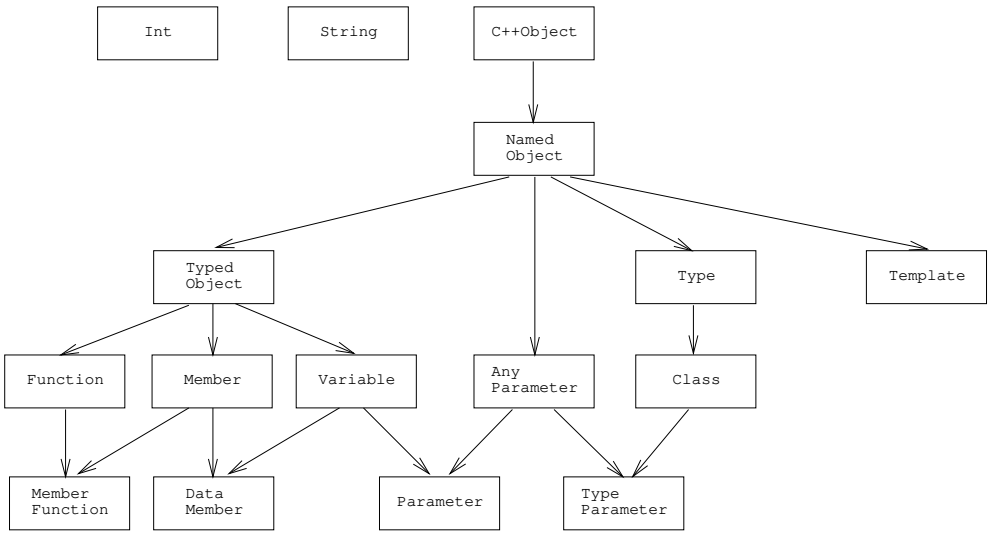

Figure 1: CCEL Class Hierarchy

need to detect [17]. lint concentrates on type mismatches and data-flow anomalies, but the stronger typing of C++ obviates the need for lint's type-checking, and data flow analysis is typically unrelated to the high-level perspective encouraged by the modular constructs of C++. Instead, C++ programmers are concerned with concepts such as the structure of a design-specific inheritance hierarchy, but lint offers no provision for design-specific error detection. The most fundamental difference, then, between lint and CCEL is that CCEL was designed from the outset to allow for the addition of *programmer-defined* constraints.

An earlier working paper describing CCEL has already been published [4], but since the time of that publication we have made significant improvements to both the language and the software architecture built around it. Such improvements include a simpler, more uniform syntax; a set of predefined types that is more expressive and is easier to understand and use; and an implementation architecture that is more portable, more robust, and has a more convenient user interface.

2 Design Considerations, Syntax, and Semantics

Our primary requirements in designing CCEL were these:

- It must offer sufficient expressive power to allow programmers to specify a wide variety of constraints, including constraints on both concrete syntax (stylistic constraints) and semantics (design and implementation constraints).
- It must be relatively easy to learn and use. In particular, the syntax and semantics of the language should mesh well with the syntax and semantics of C++. We chose as our point of departure C++ itself and the well-known assert macro.

Like the programs C++ is used to build, CCEL has an object-oriented basis. CCEL classes represent the concepts of C++. The CCEL classes are arranged in a multiple inheritance "isa" hierarchy (see Figure 1), and each class supports a particular set of member functions (see Table 1).

Syntactically, CCEL constraints resemble expressions in the predicate calculus, allowing programmers to make assertions involving existentially or universally quantified CCEL variables. In general, constraint violations are reported for each combination of CCEL variable bindings that causes an assertion to evaluate to false.

There are five parts to a CCEL constraint:

1. A unique identifer that serves as the name of the constraint.

CCEL Class	Member Function
Int	Int operator == (Int)
	Int operator < (Int)
	Int operator ! ()
	Int operator && (Int)
	Int operator || (Int)
	Int operator != (Int)
	Int operator > (Int)
	Int operator <= (Int)
	Int operator >= (Int)
String	Int operator == (String)
	Int operator < (String)
	Int matches(String)
	String operator + (String)
	Int operator != (String)
	Int operator > (String)
	Int operator <= (String)
	Int operator >= (String)
C++Object	String file()
	Int begin_line()
	Int end_line()
NamedObject	String name()
TypedObject	Type type()
	Int num_indirections()
	Int is_reference()
	Int is_static()
	Int is_volatile()
	Int is_const()
	Int is_array()
	Int is_long()
	Int is_short()
	Int is_signed()
	Int is_unsigned()
	Int is_pointer()

CCEL Class	Member Function
Type	Int has_name(String)
	Type basic_type()
	Int operator == (Type)
	Int is_convertiable_to(Type)
	Int is_enum()
	Int is_class()
	Int is_struct()
	Int is_union()
	Int is_friend(Class)
	Int is_child(Class)
	Int is_descendant(Class)
	Int is_virtual_descendant(Class)
	Int is_public_descendant(Class)
	Int operator != (Type)
Template	Int is_class_template()
	Int is_function_template()
Function	Int is_global()
	Int num_params()
	Int is_inline()
	Int is_friend(Class)
Member	Int is_private()
	Int is_protected()
	Int is_public()
Variable	Int scope_is_file()
	Int scope_is_local()
AnyParameter	Int position()
Class	
MemberFunction	Int is_virtual()
	Int is_pure_virtual()
	Int overrides(MemberFunction)
DataMember	
Parameter	Int has_default_value()
TypeParameter	

Table 1: CCEL Class Member Functions

2. A set of declarations for universally quantified CCEL variables. Such variables take as their values components of C++ programs. Each CCEL variable has a type; this type is one of the CCEL classes shown in Figure 1.

3. An assertion that comprises the essence of the constraint. Assertions may use universally quantified CCEL variables and may declare and use existentially quantified CCEL variables.

4. A scope specification that determines the region of applicability of the constraint in relation to the C++ source being checked. By default, constraints are globally applicable, but they may be restricted to a single file, function, or class.

5. A message to be issued when a violation of the constraint is detected.

Of these five parts, only the constraint identifier and the assertion are required. If omitted, the set of CCEL variables is empty, the scope of applicability is global, and constraint violations are indicated by a message in a default format.

As an example of a CCEL constraint, consider Meyers' admonition [16] that every base class in C++ should declare a virtual destructor:

```
BaseClassDtor (                          This constraint is called "BaseClassDtor":
  Class B;                               For all classes B,
  Class D | D.is_descendant(B);          for all classes D such that D is a descendant of B,
  Assert(MemberFunction B::m; |          there must exist a member function m in B such that
        m.name() == "~" + B.name() &&    m's name is a tilde followed by B's name and
        m.is_virtual());                 m is a virtual function.
  );
```

Within this constraint, the variables B and D are universally quantified,[1] and m is existentially quantified. The scope of the constraint has been omitted, so it applies to all classes. The violation message has also been omitted. This constraint is in fact an important one in practice, because programs that violate it, though legal, almost always behave incorrectly [17].

The assertion inside a constraint is evaluated only if it is possible to find a binding for each of the universally quantified variables declared in the constraint. It can therefore be useful to write constraints containing assertions that always fail, the goal being to detect the ability to bind to a universally quantified variable. For example, class templates in C++ may take either type or non-type parameters, but function templates may take only type parameters, and this inconsistency (as well as other considerations) may make it advisable to avoid non-type parameters in templates of any kind. This rule can be formalized in CCEL as follows:

```
// Templates should take only type parameters:
TypeParametersOnly (
  Template t;                            For all templates t and
  Parameter t<p>;                        all non-type parameters p of t,
  Assert(FALSE);                         issue a violation message.
);
```

Individual constraints are useful, but it is often convenient to group constraints together. This is especially the case with stylistic constraints, because a consistent style can typically be achieved only through adherence to a number of individual constraints. CCEL provides for this grouping capability through support for *constraint classes*, which comprise a set of individual constraints. For example, suppose there are several constraints on naming conventions. They could be grouped together into a constraint class called NamingConventions:

```
NamingConventions {
  // Every class name must begin with an upper case letter:
  CapitalizeClassNames (
        Class C; // C is a class, struct, or union
        Assert(C.name().matches("^[A-Z]"));
  );

  // Every function name must begin with a lower case letter:
  SmallFunctionNames (
        Function F;
        Assert(F.name().matches("^[a-z]"));
  );
};
```

Notice that the extent of constraint classes is demarcated by brackets {...}, while individual constraints use parentheses as their delimiters.

[1] B and D are both of type Class, which technically corresponds to C++ classes, structs, and unions, i.e., any language construct that may contain member functions. Hence B and D may be bound to any of these language constructs. This use of the term "class" is consistent with that employed by the C++ language reference manual [5]. Member functions in the CCEL class Class allow programmers to distinguish between classes, structs, and unions.

Sometimes what is a single conceptual constraint is best expressed using a set of simpler constraints bundled into a constraint class. The following example consists of a pair of constraints that detects undeclared assignment operators for classes that contain a pointer member or are derived from classes containing a pointer member. This is an important constraint in real-world C++ programs [16] and is one that cannot be specified in C++ itself:

```
PointersAndAssignment {
  // If a class contains a pointer member, it must declare an assignment operator:
  AssignmentMustBeDeclaredCond1 (
        Class C;
        DataMember C::cmv | cmv.is_pointer();
        Assert(MemberFunction C::cmf; | cmf.name() == "operator=");
  );

  // If a class inherits from a class containing a pointer member, the
  // derived class must declare an assignment operator:
  AssignmentMustBeDeclaredCond2 (
        Class B;
        Class D | D.is_descendant(B);
        DataMember B::bmv | bmv.is_pointer();
        Assert(MemberFunction D::dmf; | dmf.name() == "operator=");
  );
};
```

By default, a constraint applies to all code in the system. This is not always desirable. For example, a programmer might have one set of naming conventions for a class library and a different set of naming conventions for application-specific classes. CCEL explicitly provides for the need to restrict the applicability of constraints to subsets of the system being checked. In particular, the scope of a constraint may be restricted to any named portion of a C++ system: a file, a function, or a class. For files, scopes are specified in terms of UNIX shell wildcard expressions. For functions and classes, scopes are specified in terms of UNIX regular expressions. For example, if we wanted to limit the applicability of CapitalizeClassNames to the file file.h, we could declare a scope for the constraint. Such scope specifications precede the name of the constraint:

```
  // The name of every class declared in the file "file.h" must begin with a capital letter:
  File "file.h": CapitalizeClassNames (
        Class C;
        Assert(C.name().matches("^[A-Z]"));
);
```

Sometimes it is more convenient to specify where an otherwise global constraint does *not* apply. This can be accomplished by creating a new constraint with a restricted scope of application. The new constraint does nothing but *disable* the constraint that should not apply to the specified scope. Such disabling occurs through the Disable keyword. For example, to set things up so that CapitalizeClassNames applies to every C++ class *except* class X, we could disable CapitalizeClassNames for that class (note the use of a regular expression to specify only the class name "X"):

```
// Do not report violations of CapitalizeClassNames in class X:
Class "^X$" : DontCapitalizeInX (
  Disable CapitalizeClassNames;
);
```

Like individual constraints, constraint classes may be disabled. This is most frequently combined with a scoping specification:

```
// Ignore naming conventions for the file "importedFromC.h":
File "importedFromC.h" : NamingConventionsOff (
  Disable NamingConventions;
);
```

Individual members of a constraint class may be disabled by referring to them using the C++ scoping operator("::"):

```
// Turn off the constraint NamingConventions::CapitalizeClassNames for the file file.h only:
File "file.h" : SomeNamingConventionsOff (
  Disable NamingConventions::CapitalizeClassNames;
);
```

When a constraint violation is detected, a message to that effect is issued identifying the location of the CCEL constraint, the name of the C++ object bound to each universally quantified variable, and the location of each object so bound. Locations consist of a file name and a line number corresponding to the beginning of the source code for the object being identified. For example, consider this declaration for an array template:

```
template<class T, int size> class BoundedArray { ... };
```

This template violates the TypeParametersOnly constraint discussed earlier, so a violation message in the following format would be issued:

```
"constraints.ccel", line 100: TypeParametersOnly violated:
  t = BoundedArray ("BArray.h", line 15)
  p = size ("BArray.h", line 15)
```

CCEL allows this default message format to be overridden by a programmer-defined format on a per-constraint basis. Details are available in the CCEL language specification [8].

3 A Software Architecture

The architecture for our prototype constraint-checking environment is shown in Figure 2. CCEL constraints may be specified in one or more files and/or within a C++ program in the form of specially formatted comments. All features of CCEL may be used inside C++ source, but we expect programmers will use this capability primarily to specify constraints specific to a class or file, i.e. to associate a constraint with the C++ source to which it applies. For example, a constraint stating that all subclasses of a class must redefine a particular member function might be best put in the C++ source file for the class so that programmers know that they will need to define that member function. A more generic constraint, such as that every class name must begin with an upper case letter, might go in a file containing nothing but stylistic constraints.

The constraints specified in CCEL constraint files and the constraints specified in special comments in C++ source code together comprise the set of applicable constraints for the software system to be checked. This set of constraints is then used as input to a constraint checker, which employs the services of an "oracle" about the C++ system being checked. The oracle is in essence a virtual database system containing information about C++ programs. There are many actual database systems containing such information (e.g., Reprise[20], CIA++[7], and XREFDB[11]), and our virtual database interface allows us to decouple the constraint-checker from any particular database. In fact, our eventual virtual database interface will be OQL ("Object Query Language") [19], a virtual interface to *many* database systems.

The output of the constraint checker is a series of violation messages. These may be viewed as is, or they may be parsed by higher-level tools, such as Emacs [21] or the annotation editor inside FIELD [18]. Use of such tools allows programmers to see not only CCEL constraint violation diagnostics, but also the CCEL source giving rise to the violation and the C++ source that violates the constraint. This makes it much easier to locate and eliminate constraint violations.

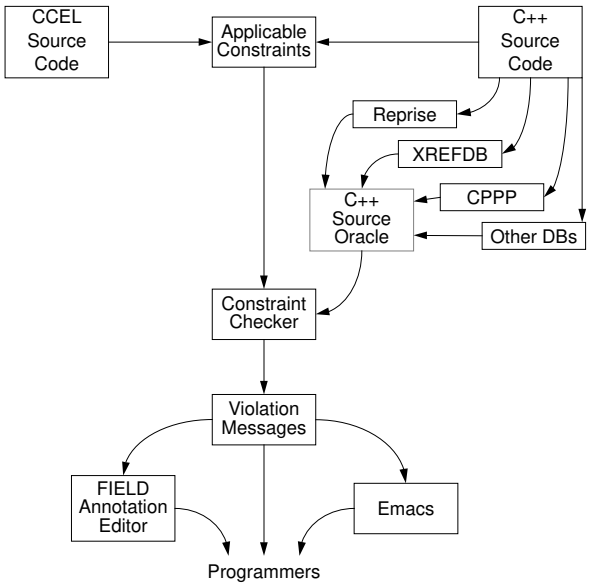

Figure 2: CCEL System Architecture

Our earlier working paper [4] described a prototype implementation of CCEL that was less elaborate than that shown in Figure 2. We are currently implementing a second-generation prototype that corresponds to the current (much expanded) language specification [8] and that is based on the database XREFDB.

4 Application to Other Languages

The classes supported by CCEL and the member functions of those classes are clearly specific to C++. The more fundamental design principles behind CCEL, however, apply equally well to other object-oriented languages. The kinds of constraints described in Section 1 exist for languages like Smalltalk and CLOS as much as they do for C++ [9]. The desireability of choosing a syntax, semantics, and conceptual model that is familiar to programmers is as important for an Eiffel constraint language as it is for CCEL. The software engineering considerations that allow CCEL constraints to be bundled into constraint classes, to be explicitly disabled, and to have user-specified scopes and violation messages are as important for Object Pascal programmers as they are for C++ software developers. Furthermore, the software architecture depicted in Figure 2 contains nothing that tailors it to the idiosyncrasies of C++. In short, the primary design considerations — both in terms of the language itself and the implementation of that language — are divorced from the specifics of C++ and can be directly applied to constraint languages for other object-oriented languages.

5 Related Work

In their analysis of the CLOS Metaobject Protocol [9], Kiczales and Lamping identified a number of issues germane to the design of extensible class libraries, and they proposed a set of informal techniques by which to specify requirements and restrictions on classes inheriting from the library. CCEL is an important step towards formalizing such requirements and restrictions and toward making them amenable to automatic verification.

Support for formal design constraints in the form of assertions or annotations was designed into Eiffel [14], has been grafted onto Ada in the language Anna [13], and has been proposed for C++ in the form of A++ [3, 2]. This work, however, has grown out of the theory of abstract data types [12], and has tended to limit itself to formally specifying the semantics of individual functions and/or collections of functions (e.g., how the member functions within a class relate to one another). In general, violations of these kinds of

constraints can only be detected at runtime. Our work on CCEL has a different focus. We are interested in constraints whose violations can be detected at compile time, and we are further interested in addressing the need to constrain relationships between classes, which Eiffel, A++, and Anna are unable to do. CCEL can also express constraints on the concrete syntax of C++ source code (e.g., CCEL class-specific naming conventions); this is also outside the purview of semantics-based constraint systems.

Acknowledgements

Yueh hong Lin provided valuable comments on earlier versions of this paper.

Support for this research was provided by the NSF under grants CCR 9111507 and CCR 9113226, by ARPA order 8225, and by ONR grant N00014-91-J-4052.

References

[1] David R. Barstow, Howard E. Shrobe, and Erik Sandewall, eds., *Interactive Programming Environments*. McGraw-Hill, 1984.

[2] Marshall P. Cline and Doug Lea, "The Behavior of C++ Classes," in *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pp. 81–91, September 1990.

[3] Marshall P. Cline and Doug Lea, "Using Annotated C++," in *Proceedings of C++ at Work - '90*, pp. 65–71, September 1990.

[4] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, "CCEL: A Metalanguage for C++," in *USENIX C++ Conference Proceedings*, August 1992. Also available as Brown University Computer Science Department Technical Report CS-92-51, October 1992.

[5] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[6] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico, *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.

[7] Judith E. Grass and Yih-Farn Chen, "The C++ Information Abstractor," in *USENIX C++ Conference Proceedings*, pp. 265–277, 1990.

[8] Yueh hong Lin and Scott Meyers, "CCEL: The C++ Constraint Expression Language." In preparation, February 1993.

[9] Gregor Kiczales and John Lamping, "Issues in the Design and Specification of Class Libraries," in *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92)* (Andreas Paepcke, ed.), pp. 435–451, October 1992.

[10] Moises Lejter, Scott Meyers, and Steven P. Reiss, "Adding Semantic Information To C++ Development Environments," in *Proceedings of C++ at Work - '90*, pp. 103–108, September 1990.

[11] Moises Lejter, Scott Meyers, and Steven P. Reiss, "Support for Maintaining Object-Oriented Programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, December 1992. Also available as Brown University Computer Science Department Technical Report CS-91-52, August 1991. An earlier version of this paper appeared in the *Proceedings of the 1991 Conference on Software Maintenance* (CSM '91), October 1991. This paper is largely drawn from two other papers [15, 10].

[12] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[13] D. Luckham, F. von Henke, B. Krieg-Bruckner, and O. Owe, *Anna, A Language for Annotating Ada Programs: Reference Manual*, vol. 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[14] Bertrand Meyer, *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, Prentice Hall, 1988.

[15] Scott Meyers, "Working with Object-Oriented Programs: The View from the Trenches is Not Always Pretty," in *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pp. 51–65, September 1990.

[16] Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.

[17] Scott Meyers and Moises Lejter, "Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++," in *USENIX C++ Conference Proceedings*, pp. 29–40, April 1991. Also available as Brown University Computer Science Department Technical Report CS-91-51, August 1991.

[18] Steven P. Reiss, "Connecting Tools using Message Passing in the FIELD Program Development Environment," *IEEE Software*, pp. 57–67, July 1990. Also available as Brown University Computer Science Department Technical Report CS-88-18, "Integration Mechanisms in the FIELD Environment," October 1988.

[19] Steven P. Reiss and Manojit Sarkar, "Generating Program Abstractions." Working paper, September 1992.

[20] David S. Rosenblum and Alexander L. Wolf, "Representing Semantically Analyzed C++ Code with Reprise," in *USENIX C++ Conference Proceedings*, pp. 119–134, April 1991.

[21] Richard M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," in *Proceedings of the ACM SIGPLAN/SIGOA Symposium on text Manipulation*, pp. 147–156, June 1981. Reprinted in [1, pp. 300–325].

A Examples

The CCEL constraints that follow serve to demonstrate not only the expressiveness of CCEL itself, but also the kinds of constraints that C++ programmers might well want to express.

The following two constraints are taken from Meyers' book [16]:

```
// Subclasses must never redefine an inherited non-virtual member function:
NoNonVirtualRedefines (
    Class B;
    Class D | D.is_descendant(B);
    MemberFunction B::bmf;
    MemberFunction D::dmf | dmf.overrides(bmf);
    Assert(bmf.is_virtual());
);

// The return type of operator= must be a reference to the class:
ReturnTypeOfAssignmentOp (
    Class C;
    MemberFunction C::mf | mf.name() == "operator=";
    Assert(mf.is_reference() && mf.type().basic_type() == C);
);
```

The constraint that structs in C++ should be the same as structs in C (useful for maintaining data structure compatibility between the two) consists of three separate constraints: (1) no struct may contain a non-public member, (2) no struct may contain a function member, and (3) no struct may have a base class. These constraints may be combined into a single constraint class `StructsAreSimple` as follows:

```
// Structs in C++ should be just like structs in C:
StructsAreSimple {

    Class s | s.is_struct();

    AllMembersPublic (
        DataMember s::mv;
        Assert(mv.is_public());
    );

    NoMemberFunctions (
        MemberFunction s::mf;
        Assert(FALSE);
    );

    NoBaseClasses (
        Class base | s.is_descendant(base);
        Assert(FALSE);
    );
};
```

Here is another constraint from Meyers' book, this one also employing a constraint class:

```
// All classes declaring or inheriting a pointer member must declare a
// copy constructor:
NecessaryCopyConstructors {

  // All classes declaring a pointer member must declare a copy ctor:
  PtrDeclImpliesCopyCtor (
      Class C;
      DataMember C::mv | mv.is_pointer();
      Assert(MemberFunction C::mf; Parameter mf(p); |
             mf.name() == C.name() && mf.num_params() == 1  &&
             p.type().basic_type() == C && p.is_reference() &&
             p.is_const());
  );

  // All classes inheriting a pointer member must declare a copy ctor:
  InherPtrImpliesCopyCtor (
      Class B;
      Class D | D.is_descendant(B);
      DataMember B::mv | mv.is_pointer();
      Assert(MemberFunction D::mf; Parameter mf(p); |
             mf.name() == D.name() && mf.num_params() == 1  &&
             p.type().basic_type() == D && p.is_reference() &&
             p.is_const());
  );
};
```

The following constraint enforces a common rule of style:

```
// Members must be declared in this order:  public, protected, private:
MemberDeclOrdering {

    Class  C;
    Member C::pub  | pub.is_public();
    Member C::prot | prot.is_protected();
    Member C::priv | priv.is_private();

    PublicBeforeProtected (
        Assert(pub.begin_line() < prot.begin_line());
    );

    PublicBeforePrivate (
        Assert(pub.begin_line() < priv.begin_line());
    );

    ProtectedBeforePrivate (
        Assert(prot.begin_line() < priv.begin_line());
    );
};
```