Appendix A

OO Programming Rules and Guidelines

OO Programming Rules and Guidelines:
Version 3

George Logothetis
Nancy Wilkinson

Software Development Technology Group

December 1, 1994

I. Introduction

This paper defines a comprehensive set of OO and C++ programming conventions to guide the writing of C++ programs. The objectives of these conventions are the following:

1. To help programmers produce code that is as easy as possible to read, review and maintain.
2. To help programmers avoid risky C++ programming practices and dark corners of the language and compiler.

The intent is to impose a discipline upon developers, over and above that imposed by the programming language, in order to enhance the comprehensibility and reliability of the code they produce. The extra effort on the part of the programmer to make information explicit in the code is deemed to be reasonable if the result will have a noticeable effect on maintainability. These conventions pertain to C++ programs. Conventions that pertain identically to both C and C++ are generally not repeated here. Exceptions to this are some of the Naming Conventions. C conventions that are important in a C++ environment in a unique way are included here, such as global namespace issues. Other C++ conventions are exactly opposite to conventions for C, e.g. using 0 rather than NULL. Where C++ conventions diverge from C, this is explicitly noted. Readers are referred to C Programming Guidelines by Thomas Plum and C Traps and Pitfalls by Andrew Koenig for basic C guidelines. The categories of conventions used in this document are rules and guidelines. Our intent is to suggest that all of these conventions should be followed by projects using C++, regardless of whether they are rules or guidelines. The difference lies in the subjectivity of the definition of compliance. The scope and use of each category is described below.

+ Rules are conventions which have a clear, non-subjective definition of compliance. Because of this, the programmer can ensure compliance on his or her own, with the help of tools and visual inspection.

+ Guidelines are conventions which have a more subjective definition of compliance. Because of this, peer review may be needed. Tools may be of help, but human judgement and experience is needed to ensure compliance.

Exceptions to any of these rules or guidelines should be rare and should be agreed upon by the project team. These conventions can be augmented by particular projects. But they may only be strengthened; that is, any convention added by a project should not violate an existing rule or guideline. New versions of this document will be issued and changes made as experience warrants. But the guiding philosophies of each category and what we consider important about conventions in general will not change.

II. Changes from Previous Version

Version 3 of this document supersedes Version 2, dated June 1. Modifications include the following:

+ The Tools section has been updated to reflect the new set of tools.

+ The Naming Conventions section has been updated slightly. Rules 6 and 7 have been strengthened; and some guidance is given about how projects can add their own project-specific naming conventions.

+ Some extra motivation and explanation has been provided for rule 25 due to changes from the C++ ANSI committee.

III. Available Tools

In order to help projects verify compliance with the following conventions, a tool has been developed to check code for compliance to a subset of the rules (about 25%). Not all rules are checkable by automated tools; some will still require human intervention to ensure compliance.

The rule checker currently checks for compliance to conventions, where possible, under the following categories:

+ Naming Conventions
+ Class Definitions
+ Canonical Class
+ Assignment Operator
+ Constructors
+ Inlines
+ Memory Management
+ Inheritance and Virtual Functions
+ Overloading
+ Hierarchy

The checker tool was implemented using a tool generator environment for C++ called gen++. The next release of the gen++ package will include the rule checker, along with a metrics gathering tool and other sample analyzers. This will give users access, not only to the checker tool, but to the environment in which to write tools of their own. This release will be a beta release, distributed to anyone interested in trying the toolkit for 6-12 months.

Gen++ currently runs on SunOS 4.1.* and hpux and will be ported to other platforms as demand dictates. If you would like to become a beta customer for the package, please send mail to mozart!c++.

IV. Sources

As input to this document, we reviewed existing sets of conventions, guidelines and rules of thumb. We pulled from these sources as was reasonable and consistent with our experience. These sources included the following:

+ Henricson and Nyquist, Programming in C++, Rules and Recommendations. Ellemtel Telecommunications Systems Laboratories. 1992.

+ J. C. Shank, "C++ Coding Standards and Guidelines for SLCÆ-2000 Access System Software Development." AT&T Bell Laboratories Internal Memorandum 54585-911115-01IM.

+ "Conventions in MacApp code." MacApp Tutorial (C++), Developer Technical Publications. Apple Computer, Inc. 1992.

+ Plum and Saks, C++ Programming Guidelines, Plum-Hall, Inc. 1991.

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

+ Robert Murray, C++ Strategies and Tactics, Addison-Wesley. 1993.

+ Jim Coplien, "Looking Over One's Shoulder at a C++ Program."

+ Timothy Budd, Object-Oriented Programming, Addison-Wesley. 1990.

+ Rebecca Wirfs-Brock, et. al., Designing Object-Oriented Software, Prentice-Hall, 1991.

+ Grady Booch, Object-Oriented Designs with Applications, Benjamin-Cummings, 1990.

V. Programming Conventions

Naming Conventions

Naming conventions are the least objective and normally most controversial of any standards. They may be somewhat arbitrary, but are very useful in increasing the readability of code. Names, such as class names, variable names, and function names can be distinguished by the reader at a glance. They can also be useful to the writer, forcing him/her to think through the role of the name as it is created.

We have tried to keep these conventions general, so that projects could impose additional naming rules that are useful in their environment and domain. In particular, a project may decide to use a prefix (separated from the name by an underscore) to provide additional information. A project could, for instance, insist that global names be prefixed with a "g" or that acronyms be provided as a prefix. We assume the following form: prefix_name (where prefix_ is optional). These conventions apply to names.

1. (G) Readable and comprehensive names should be preferred over names which are easy to type.

2. (G) Names should be as specific to the context as possible, e.g. Coordinate rather than Number.

3. (G) Names with global scope should be distinctive and unique.

4. (R) Class names should begin with an uppercase letter.

Example: String, Message, NetworkElement

5. (R) Acronyms that are part of names should be all uppercase.

6. (R) Variable names should begin with a lowercase letter.

Example: myString, msg2, flowError

7. (R) Function names should begin with a lowercase letter (except constructors and destructors).

Example: length(), execute()

8. (R) In any name which contains more than one word, the first word should follow the convention for the type of the name, and subsequent words should immediately follow, with the first letter of each word capitalized.

Example: logEvent(), PrintManager

File Organization
----

9. (R) Class definitions should be placed in class header files and class member implementations in one or more source files.

10. (G) Each header file should define one class or a small, tightly connected group of classes.

11. (G) The name of the header file should be the name of the most visible or important class being defined, suffixed by ".h".

Example: String.h, Alarm.h

12. (G) If the class name is too long, a unique and meaningful abbreviation should be chosen.

Example: NetElem.h

13. (G) Source file names should indicate the class and the most general or visible capability being defined. They should be suffixed by ".c" or ".C" (choose one and use it consistently).

Example: StrAssign.c, AlarmInit.c

Class Definitions
----

14. (R) The protection of every base class should be specifically stated.

When one class is derived from another, the default protection is private. For example:

```
class D : B { }; // D is privately derived from B
```

This is counter-intuitive for most programmers who think that the default is public inheritance. Thus, the compiler will generate a warning for constructs such as these. It is best to specifically state whether the inheritance is public, protected, or private.

15. (R) No inline function should be defined within a class definition.

Programmers should resist the temptation to define functions within class definitions. Defining them outside separates the declaration from the implementation details. It allows for the creation of separate files of inline function definitions. And it facilitates approaches to global "uninlining" of functions for debugging. It also minimizes the temptation to inline functions which should not be inline, such as constructors, destructors, and virtual functions.

16. (R) Class organization: the definition of the class should contain the following in this order:

    1. declaration of any friend functions and classes
    2. the "public:" specifier
    3. declarations of any public member functions
    4. declarations of any public data members
    5. the "protected:" specifier
    6. declarations of any protected member functions
    7. declarations of any protected data members
    8. the "private: "specifier
    9. declarations of any private member functions
    10. declarations of any private data

Class Header Files
----

17. (R) Every header file should be guarded against multiple inclusion with preprocessor directives.

Example:

```
#ifndef CLASS_H
#define CLASS_H

// header file contents ...

#endif
```

18. (R) Header files should #include header files only for classes that are directly needed by the class declaration. Otherwise, forward declarations should be used.

In general, a class is directly needed if objects of that class are used in the class declaration rather than just referenced (through a pointer or reference). Including other files needlessly increases code size. Include files needed by the class implementation will be included in the source files.

Example:

```
class Switch;
```

19. (R) Each class header file should contain the following:

    1. header file guard (as described above) at the beginning of the file
    2. a preamble C++ comment containing useful information such as file name, content, and copyright. This comment is for users of the header file.
    3. any #includes
    4. any forward declarations
    5. any extern and constant declarations
    6. any typedef definitions
    7. each class, struct, and union declaration followed by its inline function definitions

Class Source Files
----

20. (R) Source files should #include header files instead of using extern declarations.

21. (R) Each class source file should contain the following:

    1. a preamble C++ comment containing useful information, such as file name, content, and copyright. This comment is for maintainers of the source file.
    2. #includes: this will include at least the class header file for the class being defined
    3. any global data initialization
    4. any static data initialization
    5. any static function declarations
    6. any friend functions and binary operator definitions

7. member function definitions
8. any static function definitions

Reference to a Class Member
----

22. (R) Any reference to a class member within the body of a member function of that class should be prefixed by this->.

This makes it clear to the reader, at a glance, that the variable or function being referenced is a member of the class. It distinguishes a member variable from a global variable, a local variable, and most importantly, a function argument. It also distinguishes a member function from a global function and the sending of a message from a simple function call.

We feel that the use of this-> is superior to naming conventions to distinguish data members and member functions. It occurs only when the distinction is really necessary, at the use of the name in its class' member functions. In the uses of the name outside the class, for which the distinction is unnecessary, a naming convention becomes extra baggage.

However, for projects who cannot abide by this rule, an acceptable alternative would be to define additional naming conventions for data members and member functions.

Comments
----

23. (R) Comments intermingled with code should be C++-style comments (//) and C-style comments (/* */) should be used to comment out large portions of code.

This rule is meant to differentiate the use of C comments from C++ comments; and is not meant to preclude other methods for removing blocks of code, such as conditional compilation.

These rules are presented together to stress that one should not be blindly followed without the other. For example, using C comments to comment code can result in problems if C comments are also intermingled in code. If both rules are followed, the user should not have a problem with nested comments.

24. (R) All comments should be in English.

Declarations

25. (R) No declaration should appear in the initializer section of a for statement.

This rule prevents developers from using a confusing feature of C++. Things were confusing enough when the scope of such a declaration was the block surrounding the for statement. But the ANSI C++ committee has recently voted to change the scope of variables in the initializer section. It is now defined to be the header and the controlled statement.

It is hard to predict when this change will be supported by compilers. During the (long) transition period, the safest thing to do is to follow this rule and write code which is legal under the old and new rules, e.g.

```
int j;
for (j = 0; j < 10; j++) // ...
```

26. (R) Every class or struct declaration should have a name; and no declaration of a class or struct should also contain a variable declaration.

This rule forbids declarations such as

```
struct S { int i; int j; } s1;
```

27. (R) The return type of a function should be explicitly stated.

28. (R) The return type of a function (including any *'s and &'s) should appear on a line by itself above the name of the function.

This rule should be followed to allow tools to more easily find the name of the function.

29. (R) Names of formal arguments to the function should be specified and should be the same in the declaration and definition.

30. (R) In function declarations and function calls, arguments should either be all listed on one line, or each argument should be listed on its own line.

Example:

```
int
multiArgFunc(int arg1, char *arg2, String arg3);
```

or

```
int
multiArgFunc(
    int arg1,          // comment about arg1
    char *arg2,        // comment about arg2
    String arg3        // comment about arg3
);
```

Functions
----

31. (G) A function parameter should be a reference to an object instead of an object whenever possible.

Sometimes a function actually needs a copy of an object. If not, a reference to a const object should be used to avoid copying potentially large objects.

32. (G) Avoid returning non-const references or pointers to member data.

This allows direct access to class members and thus violates encapsulation and information hiding.

Use of Braces
----

33. (R) Explicit blocks with braces (even if the block is empty) should follow each if, for, while, do, and switch statement.

This rule, although it could apply to C as well as C++, is not a common part of C programming conventions, and so is stated here. In our experience, many common errors flow from incorrectly modifying code which has been written without explicit braces, especially when the null statement appears after a conditional. It increases readability as well.

Example:

```
if (...) {
}
```

instead of

```
if (...) ;
```

Canonical Class

Each time a class is created, the compiler-generated copy constructor and assignment operator should be scrutinized to determine if they are adequate for the class. If found inadequate, versions of these functions should be provided by the programmer. The following addresses when and how these functions are created.

34. (R) If objects of the class allocate subsidiary data structures on the heap or consume any other shared resource, a copy constructor and assignment operator should be explicitly provided.

In these cases, the default copy constructor and assignment operator will not suffice. The copy constructor will need to allocate more memory for the new object or explicitly access the shared resource. The assignment operator will need to free the memory or shared resource of the object which is on the left-hand side of the assignment.

Some projects might find it useful to always define these functions, even if the default is adequate. This way, the class author knows exactly what is being called. The disadvantage of this approach is that changes to the class structure necessitate changes to user-defined functions, while compiler-generated functions handle them automatically.

Assignment Operator
----

35. (R) In the assignment operator all members, and base classes, should be assigned values.

All members of an object should be explicitly assigned to in the assignment operator. Also, base class assignment operators are not automatically executed for derived class objects. Therefore, the derived class assignment operator must explicitly call the base class assignment operator.

36. (R) The assignment operator should have a check for assignment to self.

If the operands of an assignment actually refer to the same object, destroying the left-hand object before assigning it the values of the right-hand object can be disastrous. This situation should be caught at the beginning of the assignment operator function.

37. (R) The return type of the assignment operator for class T should be const T& and it should return *this.

This allows expressions of the form a = b = c to be used and to behave as the built-in assignment operator behaves. Constructors

38. (R) Constructors should use initialization lists to initialize class members rather than assign to them in the body of the constructor.

Initialization lists are generally the most efficient way to initialize object member data; in fact, they are the only way to initialize reference and const members. Initialization entails just one constructor call; but an assignment inside the body may result in a constructor call followed by a call to an assignment operator.

Example:

```
class Foo {
public:
        Foo(const char* s);
private:
        String name;
};
Foo::Foo(const char* s) : name(s) { } // String::String(const char*)

Foo::Foo(const char* s) // String::String()
{ name = s; }      // String::operator=(const char*)
```

39. (R) The members in an initialization list should be listed in the same order as they are declared inside the class.

Members are initialized in declaration order whether you list them that way or not.

Const
-----

40. (R) Functions that do not alter the values of any data member should be declared const.

This ensures that the function can be used on the widest range of objects. Not declaring a function const needlessly restricts the set of objects upon which it can be called.

41. (G) Functions that do not alter the logical state of an object should be declared const, even if that means casting away const inside the function.

>From a user's point of view, a function may not change the state of an object even if it modifies a member internally for reasons known only

to the class author. In these cases, this illusion of constness should be preserved for the user by the class author by declaring the function to be const and casting away constness inside the function where changes are made to these data members. This is much safer than leaving those casts to the user of the class. See Meyers', Effective C++, (Item 21) for a good discussion of this topic.

42. (G) Reference or pointer parameters that are not modified by the function should be declared to be references or pointers to const.

Not declaring such parameters as pointers or reference to const needlessly restricts the set of objects which can be passed to the function.

Encapsulation and Information Hiding
----

43. (G) Public and protected data members should be avoided when possible.

Public and protected data can violate encapsulation and information hiding.

44. (R) Inline accessor and mutator functions should be provided to access and modify state components.

State information should be only accessed and modified via functions. These functions should be defined as inline for efficiency reasons.

45. (R) An accessor function should not let you modify the object; thus it should return either an object, a reference to a const object or a pointer to a const object.

Returning a non-const reference or pointer to a data member would subvert the encapsulation and information hiding of the class.

Inlines

The goal is to use inlines for functions where performance is critical but to not run the risk of "outlined inlines." These are static versions of inline functions which the compiler creates in each file that calls the function, unbeknownst to the programmer. They cause code size increases without the accompanying gain in efficiency.

46. (G) Inlining should be reserved for accessor, mutator, and very short functions.

A very short function is generally 1-2 lines. Counting lines however can be complicated if one of the lines is a call to another inline function. What gets generated by the compiler for an inline function may be more than meets the eye. Other than simple forwarding functions, e.g.

```
int length(const String &s) { return s.length(); }
        // length()s are inline
```

Users should call inlines within inlines only if they understand what the compiler is likely to generate.

47. (R) Constructors, destructors, and virtual functions should not be inline.

The address of these functions may need to be preserved (even implicitly) which will result in multiple copies being created.

48. (R) Inline functions should do no I/O.

Otherwise, iostream.h must be included in the header! This causes code bloat.

NULL
----

49. (R) 0 should be used instead of NULL.

The symbol NULL is an artifact of old C when the compiler had no way to distinguish 0, the number, from 0, the pointer. In C++, this is no longer the case (except when passed to a function with ellipses arguments). NULL is not guaranteed to be compatible with all pointer types in C++. In particular, in some implementations, it is defined to be (void*)0, which cannot be assigned to all pointer types.

Note that this is counter to the C convention to use NULL instead of 0.

This rule does not mean that the literal 0 must be used; only that some constant whose value is an unadorned 0 must be used and that NULL should not be. A project-wide symbolic constant could be defined.

Memory Management
----

50. (R) new and delete should be used instead of malloc() and free().

Operators new and delete are more convenient to use than malloc() and free() and understand C++ constructs.

51. (R) delete [] should be used whenever an array is deleted.

The empty brackets are not strictly necessary when deleting arrays of objects of built-in type. They should always be used for consistency and to increase readability.

52. (R) delete should be defined whenever new has been defined.

Memory allocation and deletion should share underlying assumptions. malloc() and free() use different assumptions than new and delete.

Preprocessor

The preprocessor is not part of the language. It is used to augment the language; but its constructs and variables are unknown to the C++ compiler. Therefore, it is impossible for the compiler to process, know about, or report about anything but the transformed output. Two features of C++ can replace some uses of the preprocessor.

53. (R) const and enum should replace use of #define for manifest constants.

54. (G) Inlines and templates should replace uses of #define for macros whenever possible.

Inheritance and Virtual Functions
----

55. (R) Every base class should have a virtual destructor.

Since a derived class object can be deleted through a base class pointer, the base class must have a virtual destructor to ensure that the derived class destructor gets called.

56. (R) A derived class should not redefine a default parameter for a virtual function.

Because default parameters are statically bound while virtual functions are dynamically bound, you could end up calling the derived class function with the base class default parameter. So the 2 default parameters better be the same. See Meyers pp. 132-134 for a complete discussion of this.

57. (R) A derived class should not redefine an inherited non-virtual function.

Redefining a non-virtual function in a derived class will cause different behavior depending on how the function is called. For example, for a redefined function foo():

```
Derived der;
Derived* d = &der;
Base* b = &der;
b->foo(); // calls Base::foo()
d->foo(); // calls Der::foo()
```

Also, redefining an inherited non-virtual function violates the is-a relationship implied by inheritance.

Global Namespace

Keeping the global namespace uncluttered reduces name collisions and makes it easier to reuse other people's code. It also makes it easier to figure out where a name is coming from. In general, global variables should be used only when absolutely necessary.

58. (G) Nested classes and enums should be used when possible.

Types used only by a particular class can be declared within the scope of that class. The name is then not known outside of that one class.

59. (G) Static members should be used when possible.

Static members often provide a good alternative to global variables and functions.

60. (G) The C++ namespace feature should be used when it becomes available.

The new namespace feature of C++, which has been accepted by the ANSI committee, enables a programmer to partition the global namespace.

Code Reuse
----

61. (G) Use the Standard Component libraries where applicable.

Use of Friends
----

62. (G) Friend functions should be used only for non-member binary operators and in other exceptional cases where efficiency demands it.

Since friends violate encapsulation and information hiding, they should be generally avoided. Sometimes efficiency concerns demand that a function have direct access to the internals of a class; but this should happen very rarely. Generally, the class accessor functions can be used just as efficiently.

63. (G) Classes should only declare other classes as friends when the two are very tightly coupled.

A common example of tightly coupled classes is a container class and its iterator, e.g. List and ListIterator. Another example is a class which is part of the implementation of another class, e.g. StringRep and String. This case, however, can often be handled more effectively with a nested class.

Overloading
----

64. (G) Except for constructors, overloading should be reserved for functions that are really doing the same thing and that have unrelated argument types.

Overloading and calling overloaded functions can decrease readability because it may not be explicitly clear to the reader which version of the function is being called. Make sure the benefit of overloading a function outweighs this cost.

65. (G) Overloaded operators should be used only for classes where they make sense and in a way that mimics the built-in operators.

Overloading operators in a non-intuitive way greatly decreases the readability of C++ code. It also carries the same disadvantages of overloading non-operator functions. Remember that defining one operator is considered overloading, because built-in versions always exist.

66. (R) If an operator is defined which has a complementary operator, its complement should also be defined and in terms of the original operator.

Example:

```
int operator==(const String&, const String&);

int operator!=(const String& s1, const String& s2)
{
```

```
        return !(s1 == s2);
    }
```

67. (G) Conversion functions should be used sparingly.

Conversion functions are used implicitly by the C++ compiler. Defining too many conversions can lead to unexpected program behavior or to compile-time ambiguities. Counting on implicit conversions also decreases the readability of code. Remember that one-argument constructors are also conversion functions. See Murray's C++ Strategies and Tactics, pp. 37-44 for a good discussion of this topic.

68. (R) All unary operators should be defined as members.

This will suppress the implicit conversions on the operand, which is a good thing from the point of view of readability and comprehension. See Murray, pp. 44-47 for details.

69. (R) Compound assignments (+=, -=, *=, etc.) should be defined as members.

Since these are forms of assignment, suppression of implicit conversions on the left operand is also desirable in this case. Again, see Murray, pp. 44-47.

70. (R) All other binary operators should be defined as friends.

Conversions on the left-hand operand make sense in the case of other binary operators. Since they tend to be symmetric and conversions take place on the right, it is confusing to the user if conversions do not happen on the left as well. Again, see Murray, pp. 44-47.

Ellipses Arguments

71. (G) Ellipses should not be used except in very rare cases.
These exceptional cases should use <stdargs.h>.

Ellipses short circuit type checking and conversions. They should only be used for functions like printf() where it is impossible to predict the number and types of the arguments.

The use of <stdargs.h> for these functions is guaranteed to be portable. This replaces the use of <varargs.h> in K&R C.

Type Casting

72. (G) Type casts should not be used unless absolutely necessary.

Each occurrence of a cast should alert the reader to some special need to override the type system of C++. They should not be inserted routinely or for insignificant reasons.

References and Pointers

Using references and pointers correctly can increase the readability of C++ code by forcing a programmer to state explicitly how a variable is used.

73. (G) Pointers should be used when you want multiple references to the same object or a dynamic data structure.

In C, pointers were used to simulate "call by reference" as well as for multiple references to the same object. In C++, pointers should not be used to pass an object by reference.

74. (G) References should be used whenever a parameter is being passed "by reference".

This also leads to more readable code inside the function because parameters are not dereferenced throughout.

Templates

75. (G) The necessary member functions should be defined for T in order for T to be used as a template parameter.

Member functions of a parameterized (template) class may rely on functions of its parameter, T. For example, a sort() operation of a List<T> may assume that T has defined an operator<().

Static Object Construction
----

76. (G) Static objects from other files should not be used in functions called at static constructor time.

A well-known problem with C++ is that the order of execution of static object constructors is defined only within a single file. Therefore, references to a static object from another file with a static object constructor may fail mysteriously at the time of execution. This situation should be avoided altogether.

Initialization
----

77. (G) Initialization should be preferred to assignment.

An initialization is generally more efficient than a declaration followed by an assignment. In the case of a class object, the latter will result in two function calls (constructor plus assignment operator); but initialization entails one call to a constructor.

This implies that variables should be declared very close to where they are used. Note that this is counter to the C convention to put all declarations at the top of a block.

VI. Design Conventions

The following guidelines pertain to the creation and review of the design of an object-oriented program. Some of these can be checked and metrics about them can be collected from code (assuming a particular mapping from design to code); but their compliance must be at the design level. These are vague, as they reflect the state of object-oriented design heuristics, but may be useful as rules of thumb while creating the design.

Hierarchy

The first two guidelines caution against creating classes when other alternatives are preferable.

78. (G) Deep class hierarchies (e.g. more than 3 levels) should be scrutinized.

Deep class hierarchies increase the complexity of an application and make it more difficult to understand a particular class deep in the class structure. Sometimes, adding a function to an existing class is preferable to deriving an additional class.

79. (G) Wide class hierarchies (e.g. more than 3 children) should be scrutinized.

A wide and shallow class structure may not exploit all the commonality that exists. Wide hierarchies may also suggest the potential for the use of templates for a more compact design and implementation notation.

Class Structure
----

80. (G) A class should be sufficient and complete.

The class is sufficient if it captures enough characteristics of the abstraction to permit meaningful interactions. It is complete if it contains all meaningful characteristics.

81. (G) A class with too few (e.g. < 3) operations should be scrutinized.

Classes with too few responsibilities should sometimes be merged with other classes.

82. (G) A class with too many (e.g. > 10) operations should be scrutinized.

Classes with too much knowledge and responsibility should sometimes be split into more than one class.

83. (G) A class should be cohesive.

The operations of a class should be highly connected by data and functionality. If a class has operations which depend on disjoint sets of data members, it probably represents more than one class.

Class Relationships
----

84. (G) A class should not depend in any way on a structure of any other class (Law of Demeter).

This is a restatement of the need for encapsulation and information hiding. Classes should obtain information about other classes only through the interface of the other class.

85. (G) Class coupling should be minimized.

Class coupling is the number of collaborations a class has with another class. Fewer collaborations means that the class is less likely to be affected by changes to other parts of the system. Also, classes which do not know about many other classes may be more reusable. Operations can sometimes be reassigned or new classes created to decrease the number of collaborations.

86. (G) Templates should be used when appropriate instead of inheritance.

Use templates to describe a family of classes as a single "generic" class when the differences in structure and implementation can be characterized by a small set of type parameters.

87. (G) Inheritance should not be used for has-a relationships.

Has-a design relationships are best implemented by containment, where one class has an object (or reference or pointer to an object) of the other class as a member.

88. (G) Multiple inheritance should be used sparingly.

Multiple inheritance is a fairly new feature of object-oriented languages. Use of MI introduces complexities and potential ambiguities into any C++ program. This feature should be used only when necessary and only if all facets are well-understood.

APPENDIX A: Auditable Rules

This section provides a handful of high-level rules which provide guidance in applying the most important rules and guidelines discussed in this document.

Design Rules
----

1. Class Hierarchies are no more than 3 levels deep and 3 children wide except as required by specific applications and problem domains. Wide or deep hierarchies could be symptoms of problems (see Hierarchy).

2. Inheritance is used to capture "is-a" relationships and containment is used for "has-a" relationships between classes.

3. Each class is sufficient and complete and contains only primitive operations.

Reusability Rules
----

4. The structure and content of C++ classes adhere to the conventions of the Canonical Class, Assignment Operator, Initialization Lists, Class Definitions, Encapsulation and Information Hiding, Use of Inlines and Operator Functions sections of this document.

Classes constructed according to these conventions will be more complete and easier to reuse.

5. The global namespace is uncluttered according to the conventions described in the Global Namespace section of this document. The fewer names in the global namespace, the less likely that clashes will occur

when code is reused.

6. Programs are implemented across files adhering to the conventions described in the File Organization, Class Header Files, and Class Source Files sections of this document.

Organizing code and header files in a standard way enables widespread reuse.

7. C++ applications use available class libraries.

Using previously developed and tested reusable classes decreases development and maintenance time and increases reliability. Examples are the USL Standard Components and GDMO-defined libraries. (Note: GDMO stands for Guidelines for the Definition of Managed Objects).

Maintainability / Readability Rules
----

8. C++ names adhere to the conventions described in the Naming Conventions sections of this document.

Meaningful and consistent names enhance readability and maintainability of code by adding structure and type information into the names.

9. The form of C++ declarations and member references adhere to the conventions described in the Declarations and References to a Class Member sections of this document.

Function Point Rule
----

10. The average number of function points per class is between 1 and 4.

A function point is a software measure that defines "end-user benefit." It is expressed as the weighted sum of the following project factors: inputs, outputs, logical files, inquiries and interfaces. The resulting number is used to estimate staff effort for projects. Function points were invented by A. J. Albrecht in the mid-1970s and are described in Applied Software Measurement by Capers Jones (McGraw-Hill, 1991).

The average presented here is an attempt to predict, based on past experience with C programming, the number of function points which will map to an implementation class. No studies in the area of object-oriented programming and function points have been done. As actual measurements become available, this number must be refined and may change dramatically. For example, there are efforts to change the way function points are collected for object-oriented projects which has a class-based view. Using this approach, the number of function points per class would be much higher.

Process Rules
----

11. Chen charts are run on all class methods.

12. 10% of the staff of the project are STC-certified object architects.

An STC-certified object architect (1) has taken the STC architecture class, (2) has had training in C++, (3) has had training in OO Design, e.g. the KETC course or the STC CRC card course, and (4) has experience on an object-oriented project.