XYZ Corporation

Sample Coding Rules for C++

Contents

1 Naming Conventions

1.1 File Names

1.1.1 Filenames must be limited to the case-insensitive DOS 8.3 format, i.e. the name must have no more than 8 characters, followed by an extension with no more than 3 characters.

Exception: This rule does not apply when portability to DOS or Windows 3.1 is not an issue.

Reference: Paragraph 3.1, XYZ C++ Guidelines.

1.2 Class Names

1.2.1 Class names must begin with the prefix `XYZ`. If there is a possibility of classes with similar names being used within different business units, the prefix should be extended to indicate the business unit. The prefix must be followed by an underscore and the first letter of the class name following the underscore should be uppercase. If a class name contains more than one word, each word should begin with an uppercase letter.

Example: `class XYZMKT_AccountHistory`

Justification: The class-name prefix avoids name conflicts with names provided by class libraries (either vendor supplied or internally developed). In addition, the prefix distinguishes a class name from a member function name or variable name.

Reference: Paragraph 3.2, XYZ C++ Guidelines.

Note: CodeCheck cannot verify that each word in a class name begins with a capital letter.

1.3 Member Names

1.3.1 All private data members and private member functions must have names that start with a lower-case letter and end with an underscore.

Justification: The special spelling for private members allows them to be distinguished from protected or public members.

Reference: Paragraph 3.3, XYZ C++ Guidelines.

Note: This rule has been upgraded from a "suggested" standard to a requirement, in order to encourage its use.

2 Class Definitions

2.1 Header File Format

2.1.1 Class definitions must always be in header files. Class implementations must be in a separate source file (or files).

Justification: The class definition acts as the interface between the class and the source code that uses the class. Placing only the class definition in a header file facilitates the re-use of the class wherever it may be needed.

Reference: Paragraph 3.1, XYZ C++ Guidelines.

2.1.2 The first actual data in the header file must be a comment block which includes any information needed by the version control software as well as descriptive information about the class.

Justification: A careful description of the use and operation of a class in the comment block at the start of its header will facilitate its correct re-use in other projects.

Reference: Paragraph 4.2, XYZ C++ Guidelines.

2.1.3 Each header file must be entirely wrapped in an `#ifndef` block. The name of the wrapper macro is concatenating the header's filename and extension, separated with an underscore.

Example: Header "foo.h" is wrapped with a macro named `FOO_H:`

```
#ifndef FOO_H
#define FOO_H
//   Actual contents of file
#endif  //      FOO_H
```

Justification: Use of an #ifndef wrapper prevents multiple definition of the classes defined in the header.

Reference: Paragraph 4.1, XYZ C++ Guidelines.

2.1.4 Function definitions must not be placed in header files.

Justification: This rule encourages the strict separation of interface and implementation. In addition, when a programmer changes a function that is defined in a header file, then all source files that depend on the header have to be compiled and linked. If the function is defined in a source file, then only that source file will have to be compiled and linked.

Reference: Paragraphs 4.4 and 5.1, XYZ C++ Guidelines.

Note: This rule has been upgraded from a "suggested" standard to a requirement, in order to encourage its use. The exception for inline functions has been removed.

2.2 Public, Protected, and Private Sections

2.2.1 The public section of the class definition should come first, followed by the protected and private sections.

Justification: This rule conforms to the principle of data encapsulation.

Reference: Paragraph 4.3, XYZ C++ Guidelines.

2.2.2 All data members must be private.

Justification: This rule conforms to the principle of data encapsulation.

Reference: Paragraph 3.3, XYZ C++ Guidelines.

Note: This rule has been upgraded from a "suggested" standard to a requirement, in order to encourage its use.

2.3 Inline Functions

2.3.1 The only functions that may be inlined are trivial one-liners.

Justification: Only consider declaring inline functions where there is a known performance problem that can be ameliorated by saving the overhead of function calls. Chances are that real performance problems are due to more fundamental causes (e.g. algorithms or I/O), so don't think that inlining functions is likely to be a solution. In general there are serious drawbacks to using inline functions:

By repeating code everywhere a function is used, inline functions can dramatically expand the total amount of code being compiled.

Inline function definitions get compiled into other people's code, and cannot be changed without recompiling. This makes inline functions unsuitable for dynamic libraries, because changes in inline and formerly inline library functions are not picked up in the runtime system.

Using inline functions can greatly complicate the process of setting breakpoints in the debugger.

Reference: Paragraph 4.4.2, XYZ C++ Guidelines.

2.3.2 Do not use the inline keyword in a class declaration. Inline functions must be defined outside the class declaration, with the explicit inline keyword.

Justification: This rule enforces the strict separation of interface and implementation.

Reference: Paragraph 4.4.3, XYZ C++ Guidelines.

Note 1: This rule is automatically covered by Rule 2.1.1, which states that class member functions must be defined in a source file that is separate from the header file that contains the class definition.

Note 2: At the end of paragraph 4.4.3, the Guidelines also say, "Place inline function definitions toward the end of the .h file that includes the class declaration, or in a separate .h file." This conflicts with Rule 2.1.1.

2.4 Virtual Functions

2.4.1 The virtual keyword must be repeated in each subclass that redefines a virtual function.

Justification: For clarity in reading polymorphic code, the keyword virtual must be repeated in each subclass that redefines a virtual function, even though the keyword is not required by C++.

Reference: Paragraph 4.5, XYZ C++ Guidelines.

Note: Not enforceable with this version of CodeCheck. The new feature may be implemented by Abraxas in the near future.

2.5 Constructors and Destructors

2.5.1 Each class must have at least the following two constructors: a default constructor, and a copy constructor.

Definitions: A *default* constructor has no arguments. It is called whenever an object is declared without an explicit initializer or call to a constructor with arguments. It is also called when an array of objects is declared. The compiler will create its own default constructor for each class that does not have an explicit default constructor.

A *copy* constructor has a single argument which is a reference to another object of the same class. It is called whenever an initializer requires that the contents of one instance of a class be copied to another.

```
ObjectClass x;          // Default constructor called.
ObjectClass y = x;      // Copy constructor called.
ObjectClass y(x);               // Copy constructor called.
```

In addition, the copy constructor is called whenever a function returns an object by value. The compiler will create its own copy constructor for each class that does not have an explicit copy constructor.

Justification: If you do not define these constructors, the compiler may or may not generate what you need: do not take chances. In any case, as an object class evolves and becomes more sophisticated, you will likely need your own constructors. There are several circumstances in which the default contructor makes no sense. An object which contains a reference or pointer to another object as a data member may make no sense without the contained object. For example, in programming for graphical user interfaces you might have an object which contains a reference or pointer to a window. The object would make no sense without the window and there is no reasonable way to give the window a default value. In such a case, if you do not define a default constructor but do define the copy constructor (or any other constructor for that matter), the compiler will not define a default constructor for you.

Reference: Paragraphs 4.6-4.8, XYZ C++ Guidelines.

2.5.2 Every class must have an explicit operator=**, and then define the copy constructor in terms of** operator=.

Justification: Adherence to this rule will make explicit the exact manner in which an assignment is carried out. The following code shows the basic ambiguity:

```
ObjectClass x;
ObjectClass y;
y = x;                                  //      Calls operator=
ObjectClass z = x;      //      Calls copy constructor
```

Adherence to this rule will assure that the = sign is always interpreted in the same way.

Note: The assignment operator (operator=) is not inheritable, and you will have to define operator= explicitly for each subclass. If you do not define operator= explicitly, then depending on your compiler and the inheritance context, operator= will give you either member-by-member copies of the variables, or bitwise copies. If you do not know for sure which it will be, then your program will likely fail: do not take chances.

Reference: Paragraph 4.9, XYZ C++ Guidelines.

2.5.3 Avoid defining more than three constructors.

Justification: You can have other constructors, distinguished by argument types just like overloaded functions. However, the XYZ style is to avoid heavy constructor overloading. Define only a few simple constructors followed by initialization member function calls, rather than many constructors with lots of possible initialization arguments.

Example:

```
ObjectClass x("name", "123456", "A3", 908, 699);        // Bad.
ObjectClass y;          // Okay.
y.Name("name");
y.SSN("123456789");
y.Grade("A3");
y.NPANXX(908, 699);
```

Note: The Guidelines do not actually give a specific limit to the number of constructors. The specific limit of three was chosen as a reasonable interpretation of the intent of the Guidelines.

Reference: Paragraph 4.11, XYZ C++ Guidelines.

2.5.4 Every class must have a destructor.

Justification: If destructors have been defined consistently throughout your code, then that simple mechanism takes care of many routine housekeeping chores, and helps to prevent memory leaks.

Reference: Paragraph 4.12, XYZ C++ Guidelines.

2.5.5 If a class has a virtual **function, then its destructor must also be** virtual**.**

Justification: Failing to declare a virtual destructor could lead to calling the base class destructor instead of the destructor for a derived class object, which incompletely destroys the object and could result in memory leaks, inconsistencies, and eventual program failure.

Reference: Paragraph 4.13, XYZ C++ Guidelines.

2.6 Friends

2.6.1 Do not declare friend **functions, except for the arithmetic operators,** operator>>**, and** operator<<**.**

Justification: Code may seem simpler or more elegant if an external function or another class is given access to private or protected members of an object using a friend declaration in the class definition. However, this practice violates object encapsulation, and can break the integrity of your code.

Reference: Paragraph 4.15, XYZ C++ Guidelines.

2.6.2 Do not declare friend **classes.**

Justification: The need to declare another class as a friend is very rare indeed. Consider alternative designs before you even think about arguing for an exception to this rule.

Reference: Paragraph 4.18, XYZ C++ Guidelines.

2.7 Virtual Base Class Inheritance

2.7.1 Do not use virtual base class inheritance.

Justification: The intent of declaring a base class to be virtual is to resolve an ambiguity in multiple inheritance in which several inherited classes turn out to share a common base class: with virtual inheritance, you get one copy of the base class variables instead of multiple copies. This use of the virtual keyword has nothing whatsoever to do with virtual member functions, even though the same keyword is used. Unfortunately, the implementation of virtual base classes is tricky and error prone. The recommended XYZ style is to use single inheritance for object types, with multiple inheritance of "mixin" classes for features which should be independent.

Reference: Paragraph 4.19, XYZ C++ Guidelines.

2.8 Class Typedefs

2.8.1 Use typedef names rather than the basic C types (int, long, float, etc.) for data members.

Example: In a program dealing with money, use

```
        typedef float           Money;
        Money                           salary, bonus;  //      Good
```

rather than

```
        float                           salary, bonus;  //      Bad
```

Justification: This rule follows the principle of data axyztraction. Typedef names do not increase the type safety of the code, but they do improve its readability. Furthermore, if we decide at a later time that money should be an int or a double we only have to change the type definition, not search the code for all float declarations and then decide which ones represent money.

Reference: Paragraph 4.21, XYZ C++ Guidelines.

2.8.2 Typedef names should generally be defined within a class.

Justification: This rule follows the principle of encapsulation. If a group of classes needs the typedef name, then place the type definition in a common ancestor class. If there is no common ancestor class, then consider creating one specifically for the purpose of defining all the typedef names, constants, and enumerations that are required by the entire group.

Reference: Paragraph 4.21, XYZ C++ Guidelines.

2.9 Constants

2.9.1 A constant numerical value associated with a class must be provided by a class static member function.

Justification: Using a member function instead of a variable is consistent with the encapsulation principle, and allows for constants that are computed at runtime.

Reference: Paragraph 4.22.2, XYZ C++ Guidelines.

Note: Not enforceable with this version of CodeCheck.

2.9.2 When assigning a class object constant to a variable, do not use an access function that returns a pointer or reference to the object. Instead, use an access function that has a reference argument.

Justification: "If the constant is an object class type, then have the caller create a blank object which is passed as a reference argument to a class static member function which fills in the constant value. Do not return a pointer or a reference to a variable. In practice, the need for object constants is limited, because object constructors can be used to define initial values."

Reference: Paragraph 4.22.2, XYZ C++ Guidelines.

Note: This rule is redundant, because it is covered by 3.1.2.

2.9.3 All enumerations should be declared within a class.

Justification: Adherence to this rule ensures that every enumeration (enum) symbolic value is qualified by its class name. Thus conflicts with other enums are avoided.

Reference: Paragraph 4.22.3, XYZ C++ Guidelines.

3 Class Implementation

3.1 Member Functions

3.1.1 Do not declare a function argument to be a class type, or a pointer to a class type: use an object reference instead.

Justification: If you declare a function argument as an object class type, then a copy of the object needs to be constructed when an object is passed to the function. For larger objects in particular, that may have unexpected consequences. If you use an object pointer, then the pointer could be null, and if the function does not check that condition then the program could fail. In contrast, passing an object reference implies that the object exists before the function is called, and no copy of the object needs to made.

Reference: Paragraph 5.1.1, XYZ C++ Guidelines.

3.1.2 The declared return value for a member function or operator should normally be a C language basic type or a pointer, and not an object class type or object reference.

Justification: If the member function allocates an object (using new), then the object should be returned as a pointer, not a reference. The party responsible for later deleting the object should be conspicuously documented in a comment attached to the member function declaration. Note that returning a pointer to a new allocated object is a ripe opportunity for memory leaks and other serious reliability problems: don't do it unless there is no other way.

Exception 1: The arithmetic operators, operator=, operator>>, and operator<<.

Exception 2: A member function or operator of an object class that is intended to be simply a container may return a reference or a pointer to a contained object.

Reference: Paragraph 5.1.2, XYZ C++ Guidelines.

Note: This rule may be unenforceable if there is no lexical way to distinguish a container class from a regular class.

3.1.3 A member function or operator should not return a reference or a pointer to an internal variable.

Exception: A member function or operator of an object class that is intended simply to be a container may return a reference or a pointer to a contained object. For example, an array operator may return a reference to an array element.

Reference: Paragraph 5.1.2, XYZ C++ Guidelines.

Note: Not enforceable with this version of CodeCheck.

3.1.4 A member function must not return an object by value.

Justification: If a member function or operator needs to return an object value, have the caller provide a "blank" object as a reference argument, and have the function fill in the value. When done this way, the caller controls the creation and deletion of the object (e.g., on the stack for a temporary object), and there are fewer potential problems with memory leaks, invalid pointers, and double deletes.

Reference: Paragraph 5.1.3, XYZ C++ Guidelines.

3.1.5 Reference arguments for returning values should be at the beginning of the declared argument list.

Justification: Clarity will be improved if all returned values are on the left side, and input values are on the right.

Reference: Paragraph 5.1.3, XYZ C++ Guidelines.

3.1.6 Do not declare a member function to be const**, unless you know that, by design, all future implementations of that function will be** const**.**

Justification: Declaring a member function to be const makes a strong statement about the current and future implementations of that function.

A const function cannot change any of the variables in its own object (i.e., the "this" object). If you have a variable that is a C language basic type, then you can inspect the function definition to see that there are no assignments to data members. However, if your data members are themselves objects, then you need to check the declarations of those classes to make sure that you do not need to call any non-const member functions; otherwise, you could change a variable in one of those objects, which means that your function cannot be const. If that seems complicated, it is.

Once you declare a member function to be const, then other code will come to depend on it. You should not expect to flip-flop: once a function is declared to be const then it should stay that way forever, because dropping const is likely to entail source code changes everywhere objects of that class are used. If the function is not yet implemented, or the implementation could change, or the implementation depends on other classes that do not yet have stable interfaces, then you probably do not know enough to declare const.

If your project overuses const, then sooner or later someone will use a pointer typecast to get around an inappropriate const declaration of an object (i.e., a variable or a function argument) because they need to access non-const member functions of that object. That is undesirable because it blows a hole in the integrity of const checks. The alternative is to track down and remove inappropriate const declarations as your code evolves. The ripple effect can be serious. For example, changing a function argument from const to non-const could imply that another const function that calls it with a (previously) const argument now has to be redeclared as a non-const function, and then the const declarations in source code where that function is called have to be changed, and so forth. Spending programming time this way is not very productive.

Note: This rule should be implemented as a mild warning.

Reference: Paragraph 5.3, XYZ C++ Guidelines.

3.1.7 Do not declare function arguments that are object references or object pointers to be const.

Justification: The essential meaning of const applied to an object reference or an object pointer is that you can only use const member functions of that object. However, the caller should not have to know whether or not an object changes an encapsulated variable somewhere during the execution of a member function call. If the public interface of the argument object has only non-const member functions, then if you declare the argument to be const, you can't do anything with it.

Reference: Paragraph 5.4.3, XYZ C++ Guidelines.

4 Global Rules

4.1 Comments

4.1.1 Comments must use the C++ style "//..." rather than the C style "/* ... */".

Justification: Adherence to this rule makes it easy to comment out a large section of code for testing purposes by temporarily including it within a C style comment or #if block.

Reference: Paragraph 4.2, XYZ C++ Guidelines.

4.2 Constants

4.2.1 Do not define global constants as macros.

Justification: A global constant defined as a macro is not subject to type checking by the compiler. It is safer to define a global constant as a const variable with file scope, initialized at compile time.

Example:

```
#define kMaxDefs     300            //      Bad.
const int kMaxDefs = 300;       //      Good.
```

Note: The Guidelines refer to "extern const variables," but there are no such things in C++. The author must have meant "const variables with file scope." This is true because const variables cannot have external linkage in C++, although they do in ordinary C. In effect, C++ const variables are static, i.e. not seen by the linker, and cannot be external (visible to the linker).

Reference: Paragraphs 4.22.1 and 4.22.4, XYZ C++ Guidelines.

4.2.2 Every enumeration must be given an enum type name.

Justification: There is a large class of bugs that can be prevented by using explicit enum type names in the declaration of all variables and function parameters that can be assigned an enum constant value.

Reference: Paragraph 4.22.3, XYZ C++ Guidelines.

4.3 Variables

4.3.1 Do not use global variables.

Justification: Global variables create the opportunity for unintended and unexpected side-effects when code is altered. Static members should be used rather than global variables whenever possible. One exception: use of static members can cause concurrency problems when you are working in a multi-threaded environment.

Reference: Paragraphs 4.20 and 4.22.5, XYZ C++ Guidelines.

4.4 Functions

4.4.1 The return value of a function should always be stated explicitly in the declaration.

Justification: The implied int return type for a function that does not have an explicit return type is an oxyzolete feature of C++.

Reference: Paragraph 4.4, XYZ C++ Guidelines.

4.4.2 Never define a global binary operator whose arguments are two different class types. Instead, define such operators as member functions of the class of the first argument.

Justification: Adherence to this rule avoids the use of friend global operators (see Rule 2.7.1 concerning friends). The recommended style is just as good, and is consistent with SmallTalk.

Reference: Paragraph 4.16, XYZ C++ Guidelines.

5 Enforcing Standards with CodeCheck

5.1 Overview

CodeCheck is a tool for professional programmers which detects violations of standards for C++ source code. It works very much like a compiler, in that it reads parses and understands every line of code, including lines in header files, exactly the way a compiler does. However, instead of compiling the source code into machine code, CodeCheck detects violations of a specified set of coding rules, and issues warning messages and advisories. In other words, CodeCheck performs an automated code walk-through, using a set of coding rules to determine when the standards have been violated.

CodeCheck is a command-line tool, similar in operation to most Unix, DOS, and VMS tools. Windows and OS/2-PM programmers must use a DOS shell to run CodeCheck, while Macintosh programmers must use the MPW shell.

In order to find the header files that each source file includes, CodeCheck must know which directories to search. In most cases, the CodeCheck user lists these directories in an environmental variable named INCLUDE. Please see the CodeCheck Technical Notes for discussion and examples of environmental variables for each operating system.

In a typical application of CodeCheck, (using the XYZ Coding Rules, for example), the CodeCheck user only wishes to apply to coding rules to the source and header files that comprise the current project, and does not want to apply these rules to "standard" C and C++ headers, nor to libraries that have already been thoroughly tested. In this case, once again, CodeCheck needs to know which directories contain source and header files that are to be checked, and which directories are to be excluded from checking. The directories that are to be excluded are listed in the environmental variable CCEXCLUDE. Again, please see the CodeCheck Technical Notes for discussion and examples of environmental variables for each operating system. *Coding rules will be applied to all source and header files in any directory not listed in this environmental variable.*

In order to find the CodeCheck rule files that the user might wish to apply, CodeCheck must also know which directories to search for rule files. These are listed in the environmental variable named CCRULES. Please see the CodeCheck Technical Notes for discussion and examples of environmental variables for each operating system.

5.2 CodeCheck Messages and Options

CodeCheck sends all advisory messages to the stderr output stream. Each message includes the file name and line number of the line that contained the problematic code. To see these messages in the exact context in which they occurred, have CodeCheck generate a listing file (use the **-L** option). The listing file will show each advisory message immediately underneath the offending line of code, with a marker pointing to the token that was being parsed when the message was triggered.

Most professional-grade C and C++ code makes extensive use of the preprocessor to selectively suppress platform-specific and debugging code. The CodeCheck listing file clearly distinguishes suppressed code from live code, by simply omitting line numbers on lines that are suppressed by the preprocessor. This can be extremely useful when trying to determine which code the compiler actually sees when it parses the source code.

To have CodeCheck display all preprocessor macros completely expanded in the listing file, use the **-M** option. When this option is in force, every line that contains a macro will be shown twice: first before expansion, and second after all macros have been expanded. This feature can be invaluable when debugging complex or deeply nested macros.

To have CodeCheck display all header files in the listing file, use the **-H** option. Although this may create huge listing files, it is essential when trying to debug C++ class libraries (which are normally defined in headers).

Alone among all command-line shells, MS-DOS does not allow the programmer to redirect the stderr output stream to a file. For deprived DOS users, CodeCheck provides a special option (**-O**), which causes CodeCheck to send all advisory and error messages to a file named "stderr.out" instead.

Assuming that all CodeCheck users will use the INCLUDE and CCEXCLUDE environmental variables to control which source and header files are read and checked, the -S3 option should always be specified. This option tells CodeCheck to check every header file, except those in specifically excluded directories.

5.3 Special Keywords and C++ Dialects

CodeCheck can handle all of the common C and C++ dialects, as implemented by the major compiler vendors. Each of these dialects differs from standard C or C++, usually by virtue of having its own special reserved keywords (*e.g.* near, far, huge) and predefined macro constants (*e.g.* __BORLANDC__). Many dialects also have language extensions, in the form of special grammatical constructions that are not allowed in ANSI standard C or C++. Because of these dialectical variations, CodeCheck needs to know which compiler is being used to compile the source code. *This is true even if the source code is pure ANSI standard, because the compiler vendor's header files usually make use of these extensions*. The **-K** option informs CodeCheck which compiler the source code is intended for.

In addition, the user can predefine macro constants on the command line, with the **-D** option. This option works in exactly the same way for CodeCheck as it does for the major compilers.

5.4 Suggested CodeCheck Options for XYZ

First, remember to set the INCLUDE and CCEXCLUDE environmental variables. Without this information, CodeCheck cannot function properly. Set INCLUDE to point to the directories containing the header files associated with the C or C++ compiler in current use. Normally, CCEXCLUDE will also be set to these same directories, to prevent CodeCheck from applying coding standards to headers located in these directories.

For a project that uses Microsoft C++ on a DOS or Windows machine, when a full listing file is desired, the following command line will cause CodeCheck to apply the XYZ coding rules to all source files in the current directory:chk32 -Rxyz -K7 -L -H -S3 *.c

The option **-K7** identifies the dialect used by all Microsoft C++ compilers. For a project that uses generic (AT&T) C++ on a Unix machine, only the **-K** option changes:check -Rxyz -K4 -L -H -S3 *.c

For a project that uses Borland C++ on an OS/2 machine, once again only the **-K** option changes:check -Rxyz -K6 -L -H -S3 *.c

As an alternative to the wildcard file specification in the above examples, it is possible to list explicitly every file that is to be checked, or to list the name of a file that simply contains the name of every file that is to be checked. The latter is called a "project file", and must have the extension ".ccp". Common command-line options can also be placed at the beginning of the project file, one per line. See the CodeCheck Reference Manual for details about project files.

6 CodeCheck Rule File

The following contains a complete reference to the actual rule file used to implement the XYZ C++ coding standards. Please see the file Master.doc for complete list of CodeCheck triggers.

Sample Rule File for XYZ

7 CodeCheck Listing File

The listing file generated by CodeCheck while applying the XYZ rules to the test source file xyz_test.c follows. Note that every warning is shown with a pointer that indicates the token that CodeCheck was parsing when it detected the problem.

Sample Listing File

8 CodeCheck Notes for using XYZ Rule File

The XYZ Notes contain advanced information on using sample rule file in a working environment.

9 Bibliography

Cobb Loren. **C and C++ Source Code Analysis Using CodeCheck**. Portland, Oregon: Abraxas Software, 1995.

Cobb, Loren. **CodeCheck Reference**. Portland, Oregon: Abraxas Software, 1995.

Ellis, Margaret A., and Stroustrup, Bjarne. **The Annotated C++ Reference Manual**. Reading, Mass.: Addison-Wesley Publishing Co., 1990.