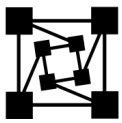
ParaSoft®
codewizard®

Programming Effectively in C++



ParaSoft®

Introduction

Errors don't just occur; they appear in code because someone puts them there. Errors are not an unavoidable part of life, nor are they an unavoidable part of coding. One of the most effective ways to reduce errors isn't through convoluted testing and debugging: it's through enforcing coding standards that will prevent errors from occurring in the first place.

The software industry is only the latest to encounter quality control problems. American auto manufacturers such as General Motors were plagued with quality problems during the 1970s. Manufacturers assumed that defects were an inevitable result of production, so when they tried to improve product quality, they didn't take any steps to reduce the number of defects caused. Instead, they increased their effort to remove defects during the later stages of production. The increased quality control forces caught more defects, but the steady stream of defects continued. These manufacturers ultimately failed to significantly reduce the number of product defects.

Japanese auto manufacturers, on the other hand, took a different (and more successful) approach. Instead of trying to improve product quality by fixing defects at the end of the production line, they made an effort to prevent defects from occurring in the first place. They determined where, how, when, and why defects were being introduced into cars. They then devised strategies that would prevent workers from introducing the defects into the cars. This innovative approach of preventing errors led to a dramatic decrease in automobile defects.

The Japanese auto manufacturers found that the secret to preventing errors was reducing the opportunity for making errors. This same principle can and should be applied to the software industry.

The current practice in software development is to produce code as rapidly as possible, then extensively test and debug it. This strategy is dangerous for two reasons:

1. Debugging is costly, time-consuming, and often delays releases or causes projects to run over budget.

2. Extensive debugging does not always find and fix all of the errors that were introduced into the code.

The key to producing higher quality software is to stop assuming that errors are an inevitable result of coding and start focusing on preventing errors from entering code in the first place. Error prevention isn't such a daunting task if you can find a way to enforce coding standards consistently and automatically.

Coding standards are language-specific “rules” that, if followed, will significantly reduce the opportunity for you to introduce errors into an application. Coding standards should be implemented in all languages, including C, C++, Java, Visual Basic, Cobol, Ada, and HTML.

Coding standards are especially critical for developers who work with object-oriented languages such as C++. Today’s object-oriented languages can be a double-edged sword for professional developers. They possess many new and powerful capabilities, but at the same time open the door to a host of new programming errors. As a result, developing programs that are effective, portable, and bug-free has become a greater challenge than ever before. This is especially true of C++, a rich and powerful language that many developers have had difficulty mastering.

An extensible language, C++ provides developers with a wide range of object-oriented features. For example, all new data types that are characterized from their inherent counterparts can be determined in a more flexible manner. Support for templates (parameterized types) is available and constructs for managing exceptions are in the language specification.

Although C++ was developed as an extension of C, there are many dangers to consider when moving code and coding techniques from C to C++. Many C tricks cannot be safely or effectively applied to C++ programming.

C++ can be simple, productive, and almost flawless when used with a group of classes that are rationally selected and meticulously crafted. Such classes should be able to automatically manipulate memory management, aliasing, initialization, and clean-up, type conversions, and other complexities that often afflict developers.

C++ can be extremely gratifying when used correctly. It can also be extremely frustrating when used improperly. With so many possibilities available, the biggest hurdle facing C++ developers is often learning what to do and when to do it.

When designing a C++ program, you should ask yourself questions such as:

- What is the proper return for the assignment operator?
- How should operator `new` behave when it can’t find enough memory?
- How do I implement a class constant?
- How should I write a member initialization list?

Knowing what you want to do is only half the battle. Knowing the best way to implement your plans is the critical step. If the consequences of various solutions are not closely examined, the resulting program may perform in an unpredictable and even baffling manner. Such design flaws may pass even the most stringent quality assurance tests. There

might be a great number of bugs hiding in your production code that don't surface until they create chaos for a customer.

Because of these complications, C++ developers can find themselves spending hours and hours studying just a few lines of code, trying to understand what they really do. Out of this frustration, an oral tradition has begun where C++ ideas and tricks are passed from developer to developer. This programming knowledge has been gathered and reproduced by ParaSoft in a product called CodeWizard.

CodeWizard

CodeWizard is an automatic source code analysis tool based on programming ideas from C and C++ experts. CodeWizard enforces over 150 built-in C/C++ coding standards designed to prevent errors. These standards are highly sophisticated and cannot be enforced by compilers. CodeWizard manages the correlations between complicated coding standards and the minimal information used by compilers across the entire class hierarchy. CodeWizard implements types of sophisticated C/C++ coding standards that other tools cannot--standards that are non-trivial and do not duplicate compiler warnings.

CodeWizard can also be customized to automatically enforce custom coding standards. Custom coding standards are rules that are specific to a certain development team, or even a certain developer. There are two main types of custom coding standards: company coding standards and personal coding standards. Company coding standards are rules that are specific to your company or development team. For example, a rule that enforces a naming convention unique to your company would be a company coding standard. Personal coding standards are rules that help you prevent your most common errors. Every time you make an error, you should determine why it occurred, then design a personal coding standard that prevents it from reoccurring. If you do this religiously, you will quickly build a set of coding standards that can prevent your most common errors from entering your code.

If you develop software for embedded systems, CodeWizard can help you prevent errors before your code ever leaves the host system. CodeWizard enforces a set of coding standards designed specifically to prevent errors in embedded software development; it also lets you easily create and enforce your own embedded development coding standards (for example, rules for software designed for a certain type of PDA or smart phone). For more information on CodeWizard's support for embedded software development, see our paper "Strategies for Preventing and Detecting Errors in Embedded Software Development."

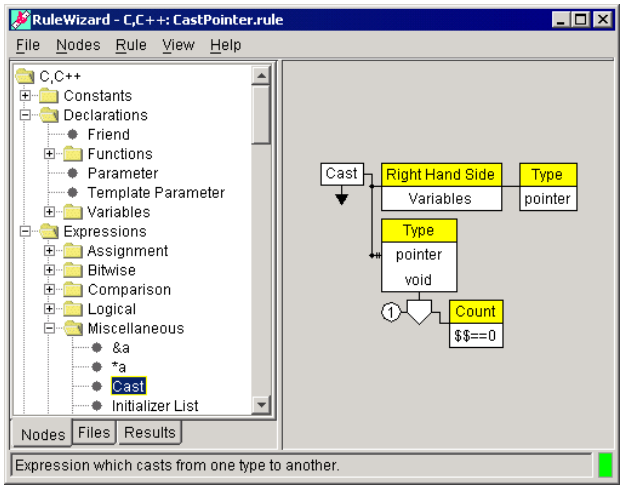
Enforcing Coding Standards

CodeWizard encourages you to use logical and portable C and C++. Only language features that are in the ANSI/ISO standard are recommended. This may sometimes seem arti-

ficially restrictive, but it is important. To illustrate, an example on More Effective C++ Item 22 examines the case of passing and returning objects by reference instead of by value. Because the behavior of passing an object by value is determined by the object's copy constructor, it can be an expensive operation. Your function may use an object as a parameter as well as a return value. In this case, both objects' copy constructors are called. When the function returns, both objects' destructors are called as well. If your object has inherited objects with their own constructors and destructors, they must be called as well. You can quickly end up calling many more constructors and destructors than you might have expected. If you pass your arguments by reference instead, no constructors or destructors are called because no new objects are created.

CodeWizard emphasizes the programming mistakes that the native compiler commonly overlooks. Compilers are designed to generate code, and therefore do not perform as deep and thorough checking as CodeWizard does. CodeWizard helps you avoid problems that the compiler is not able to detect.

In addition, CodeWizard's RuleWizard feature lets you design custom rules by graphically expressing the pattern that you want CodeWizard to look for during automatic coding standard enforcement. Rules are created by selecting a main "node," then adding additional elements until the rule expresses the pattern that you want to check for. Rule elements are added by pointing, clicking, and entering values into dialog boxes. You can also use this tool to customize many of CodeWizard's built-in coding standards.



The RuleWizard GUI

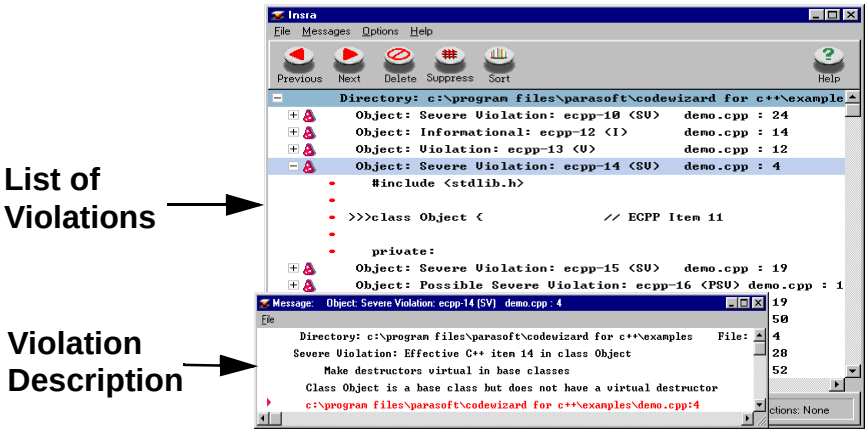
Violation Messages

CodeWizard comes with an easy-to-use GUI for UNIX platforms and installs directly into

Microsoft Visual C++ in Windows systems. Whenever an item violation is found, CodeWizard produces a violation message that tells you which item was violated and explains the violation. CodeWizard can also display the code responsible for the violation, including the line number.

Because coding standards are rules of thumb, violations will sometimes be reported that are not relevant to a particular situation. That's why CodeWizard includes flexible violation suppression features. This allows CodeWizard to learn your preferences, overcoming the shortcomings of programs like Lint. You can even save suppression options to a separate file, effectively customizing CodeWizard for a particular project or user.

Problems detected by CodeWizard indicate hazards in your code that can lead to heap corruption, dangling pointers, ambiguous initializations, and dependencies on compiler/linker behavior. These violations can cause maintenance and portability problems, but can be easily regulated if caught early in the development cycle. By using CodeWizard regularly throughout a project, you will not only improve your code but actually reduce the time spent on future development, maintenance, and porting.



The CodeWizard GUI

Using CodeWizard

The earlier CodeWizard is used in the development process, the greater the benefits. During the typical cycle of code> compile> link> test, you should use CodeWizard to augment the compile phase. Many problems will be caught before the program is even linked.

If you follow the guidelines suggested by CodeWizard, the most common C and C++ pit-

falls can be hurdled. Ultimately, CodeWizard can help you to better understand how C and C++ work, why these languages work in this manner, and how to use C and C++ to your advantage.

CodeWizard is easy to use at any development stage. Descriptions of how it can be used at various stages can be found in the following sections.

Design and Early Coding

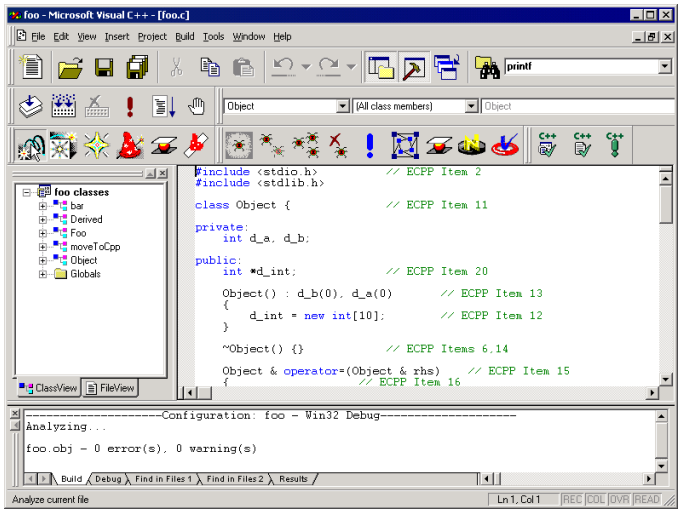
During the design and early coding stage, use CodeWizard to check new code that you are writing in a specific file. Let us assume that the file to be checked is called foo.cc. You would normally compile it with:

```
CC -c foo.cc
```

To run CodeWizard, simply type:

```
codewizard -c foo.cc
```

CodeWizard will check the program, print any violation messages to the Insra GUI, and then pass the code to your regular compiler. Because the object code is generated, there is no need to repeat the compilation process. Windows users can check a single file using the `codewizard` command, or by opening that file in Visual C++, then clicking the **Analyze File** tool bar button.



CodeWizard Integrates with Visual C++

Mid-Project Development

During advanced stages of development, when the code is compiled using make or other similar utilities, simply replace the name of the compiler in your makefile with `codewizard`.

```
make CC=codewizard
```

CodeWizard will examine the files, print the violation messages, and then convey the files to the compiler for code generation. Windows users can achieve the same effect by replacing the call to `cl` with a call to `codewizard`, or by opening their project in Developer Studio, then clicking either the **Analyze All** or **Analyze Modified** tool bar button.

Checking Without Compiling

CodeWizard can also be run as a separate pass by setting an extra option. In this case, CodeWizard would generate reports of violations, but not pass the file to the compiler.

Conclusion

As CodeWizard is used throughout the development process, you will quickly discover how essential it is in producing quality C/C++ code. By automatically enforcing coding standards, CodeWizard will make your code more readable, extensible, and maintainable with negligible overhead. When integrated into the development cycle, CodeWizard will automatically prevent errors, which improves software quality at the same time that it reduces development time and cost.

Availability

CodeWizard is available at www.parasoft.com. To learn more about how CodeWizard and other ParaSoft development tools can help your department prevent and detect errors, talk to a Software Quality Specialist today at 1-888-305-0041, or visit www.parasoft.com.

CodeWizard Item Summary

Severity	Explanation
I	Informational
PV	Possible Violation
V	Violation
PSV	Possible Severe Violation
PV	Severe Violation

Effective C++ Items

Category	Item	Description	Severity
Shifting from C to C++	2	Prefer `iostream.h` to `stdio.h`	I
	3	Use `new` and `delete` instead of `malloc` and `free`	V
	5	Use the same form in corresponding calls to `new` and `delete`	V
	6	Call `delete` on pointer members in destructors	SV/V
Memory Management	7	Check the return value of `new`	I
	8	Adhere to convention when writing `new`	V
	9	Avoid hiding the global `new`	SV
	10	Write `delete` if you write `new`	SV
	11	Define a copy constructor and assignment operator for classes with dynamically allocated memory	V/SV
	12	Prefer initialization to assignment in constructors	I

Category	Item	Description	Severity
Constructors, Destructors, and Assignment Operators	13	List members in an initialization list in the order in which they are declared	V
	14	Make destructors virtual in base classes	I/SV
	15	Have `operator=` return a reference to `*this`	SV
	16	Assign to all data members in `operator=`	PSV
	17	Check for assignment to self in `operator=`	PV
Classes and Functions: Design and Declaration	19	Differentiate among member functions, global functions, and friend functions	V
	20	Avoid data members in the public interface	V
	22	Pass and return objects by reference instead of by value	V
	23	Don’t try to return a reference when you must return an object	PSV
	25	Avoid overloading on a pointer and a numerical type	V
	29	Avoid returning "handles" to internal data from const member functions	SV
Classes and Functions: Implementation	30	Avoid member functions that return pointers or references to members less accessible than themselves	V
	31	Never return a reference to a local object or a dereferenced pointer initialized by `new` within the function	PSV/SV

Category	Item	Description	Severity
Inheritance and Object-Oriented Design	37	Never redefine an inherited non-virtual function	PV/V
	38	Never redefine an inherited default parameter value	I/V
	39	Avoid casts down the inheritance hierarchy	I

More Effective C++ Items

Category	Item	Description	Severity
Basics	2	Prefer C++-style casts	V
	5	Be wary of user-defined conversion functions	I/V
Operators	6	Distinguish between prefix and postfix forms of increment and decrement operators	V
	7	Never overload &&, ||, or ,	V
	22	Consider using `op=` instead of stand-alone `op`	V
Efficiency	24	Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI	V
Techniques	26	Limiting the number of objects of a class	I

Meyers-Klaus Items

Category	Item	Description	Severity
Constructors/ Deconstructors Assignment	13	Avoid calling virtual functions from constructors and destructors	SV
Implementation	23	Avoid using "…" in function parameter lists	I

Universal Coding Standard Items

Item	Description	Severity
2	Do not declare protected data members	V
3	Do not declare the constructor or destructor to be inline	V
4	Declare at least one constructor to prevent the compiler from doing so	V
5	Pointers to functions should use a typedef	V
6	Never convert a const to a non-const	SV
7	Do not use the `?:` operator	V
8	Each class must declare the public, protected, and private sections in that order	V
9	In the public section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and others	V
10	In the protected section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and others	V
11	In the private section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and others	V
12	If a function has no parameters, use () instead of `(void)`	V
13	If, else, while, and do statements shall be followed by a block, even if it is empty	V
14	If a block is a single statement, enclose it in braces	V
15	Whenever a global variable or function is used, use the :: operator	I
16	Do not use public data members	V
17	If a function has any virtual functions it shall have a virtual destructor	SV

Item	Description	Severity
18	Public member functions shall return const handles to member data	SV
19	A class that has pointer members shall have an `operator=` and a copy constructor	V
20	If a subclass implements a virtual function, use the virtual keyword	V
21	Member functions shall not be defined in the class definition	V
22	Ellipses shall not be used	V
23	Functions shall explicitly declare their return types	V
24	A pointer to a class shall not be converted to a pointer of a second class unless it inherits from the second	SV
25	A pointer to an abstract class shall not be converted to a pointer that inherits from that class	SV
26	Do not use the friend mechanism	V
27	When working with float or double values, use `<=` and `=>` instead of `==`	V
28	Do not overload functions within a template class	SV
29	Do not define structs that contain member function	V
30	Do not directly access global data from a constructor	V
31	Do not use multiple inheritance	V
32	Initialize all variables	V
33	All pointers should be initialized to zero	V
34	Always terminate a case statement with `break`	V
35	Always provide a default branch for switch statements	V
36	Do not use the goto statement	V
37	Provide only one return statement in a function	I

User Items

Item	Description	Severity	C/C++
ArrayElementAccess	Array elements shall be accessed by the array operator [ ]	V	C/C++
AssignAllMemberVar	Assign to all member variables in `operator=` functions	V	C++
AssignCharTooHigh	Do not use constants that are greater than the char type's legal range	V	C/C++
AssignCharTooLow	Do not use constants that are less than the char type's legal range	V	C/C++
AssignUnCharTooHigh	Do not use constants that are greater than the unsigned char type's legal range	V	C/C++
AssignUnCharTooLow	Do not use constants that are less than the unsigned char type's legal range	V	C/C++
AssignmentOperator	Declare an assignment operator for each class with pointer member variables	V	C++
BaseDestructors	Make destructors virtual for all base classes	V	C++
BitwiseInCondition	Do not use the bitwise operator in conditionals	PV	C/C++
BreakInForLoop	Avoid breaks in for loops	SV	C/C++
CastFuncPtrToPrimPtr	Do not cast a pointer to a function to a pointer of primitive type	V	C/C++
CastPointer	Do not cast pointers to non-pointers	V	C/C++

Item	Description	Severity	C/C++
CastUnsigned	Do not cast an unsigned pointer to an unsigned int	V	C/C++
CharCompare	Do not compare chars to constants out of char range	SV	C/C++
CharacterTest	Use the ctype.h facilities for character test	PV	C/C++
CommaOperator	The comma operator shall only be used in for statements and variable declarations	V	C/C++
ConstParam	Declare reference parameters as const reference whenever possible	V	C/C++
ConstPointerFunction Call	Const data type should be used for pointers in function calls if the pointer will not be modified	V	C/C++
DeclDimArray	Don't declare the magnitude of an array declaration	PV	C/C++
DeclareArray Magnitude	Don't declare the magnitude of a single dimensional array in the parameter declaration	V	C/C++
DeclareBitfield	Do not declare member variables as bitfields	I	C/C++
DeclareExplicit Constructor	Do not use the keyword 'explicit' for a constructor	I	C/C++
DeclareMutable	Do not declare member variables with the 'mutable' keyword	I	C++
DeclareRegister	Do not declare local variables with the 'register' keyword	I	C/C++

Item	Description	Severity	C/C++
DeclareStaticLocal	Do not declare local variables with the 'static' keyword	I	C/C++
DeleteIfNew	Write operator `delete` if you write operator `new`	V	C++
DeleteNonPointer	Do not call `delete` on a non-pointer	SV	C++
DoWhile	Avoid do statements	I	C/C++
EnumKeyword	Do not use an enum keyword to declare a variable in C++	I	C++
EnumNaming Convention	In an enumerated list, list members (elements) shall be in uppercase and names or tags for the list shall be in lowercase	PV	C/C++
EqualityFloat	Do not check floats for equality; check for greater than or less than	SV	C/C++
ExplicitEnumValues	When using enum, the values of each member should be explicitly declared	PV	C/C++
ExplicitLogicalTest	Use explicit logical tests in conditional expressions	V	C/C++
ExprInSizeof	Avoid expressions in `sizeof` operator	V	C/C++
FileNameConvention	Use lowercase for file names	PV	C/C++
ForLoopVarAssign	Do not assign to loop control variables in the body of a for loop	V	C/C++
FractionLoss	Do not assign the dividend of 2 ints to a float	V	C/C++
FuncModifyGlobalVar	Avoid functions that modify the global variable	V	C/C++

Item	Description	Severity	C/C++
FunctionSize	Avoid functions with over 50 lines	I	C/C++
GlobalPrefixExclude	Only use global prefixes for global variables	I	C/C++
GlobalVarFound	Avoid global variables	I	C/C++
HeaderInitialization	Headers should not contain any initialization	PV	C/C++
IfAssign	Avoid assignment in if statement condition	SV	C/C++
IfElse	Give each if statements an else clause	I	C/C++
ImplicitUnsignedInit	Do not initialize unsigned integer variables with signed constants	V	C/C++
InitCharOutOfRange	Avoid constants out of range for the char type	V	C/C++
InitPointerVar	Initialize all pointer variables	V	C/C++
InitUnCharOutOf Range	Avoid constants out of range for the unsigned char type	V	C/C++
LocalVariableNames	Local variable names shall be proper lowercase	PV	C/C++
LongConst	Use capital “L” instead of lowercase “l” to indicate long	SV	C/C++
ManyCases	Avoid switch statements with many cases	I	C/C++
MetricBlockofCode	Number of blocks of code per function	V	C++
MetricBreakEncap	Number of global variable references per member function	V	C++

Item	Description	Severity	C/C++
MetricFuncCall	Number of function calls per function	V	C++
MetricInheritance	Number of base classes	V	C++
MetricMembers	Number of data members per class	V	C++
MetricMethod	Number of methods per class	V	C++
MetricParam	Number of parameters per method	V	C++
MetricPrivateMembers	Number of private data members per class	V	C++
MetricPrivateMethod	Number of private methods per class	V	C++
MetricProtected Members	Number of protected data members per class	V	C++
MetricProtectedMethod	Number of protected methods per class	V	C++
MetricPublicMembers	Number of public data members per class	V	C++
MetricPublicMethod	Number of public methods per class	V	C++
ModifyInCondition	Do not use operator ++ or -- in the conditional part of if, while, or switch	PV	C/C++
NameBool	Begin boolean type variable names with ‘b’	I	C/C++
NameClass	Begin class names with an uppercase letter	I	C++
NameConflict	Avoid internal or external name conflict	V	C
NameConstantVar	Begin constant variable names with ‘c’	I	C/C++

Item	Description	Severity	C/C++
NameDataMember	Begin class data member names with ‘its’	I	C++
NameDouble	Begin double type variable names with ‘d’	I	C/C++
NameEnumType	Begin enumerated type names with an uppercase letter that is prefixed by the software element and suffixed by ‘_t_’	I	C/C++
NameFloat	Begin float type variable names with ‘f’	I	C/C++
NameFunction	Begin function names with an uppercase letter	V	C++
NameGlobalVar	Begin global variable names with ‘the’	I	C/C++
NameInt	Begin integer names with ‘i’	I	C/C++
NameIsFunction	Begin ‘is’ function names with bool values	I	C/C++
NameLongInt	Begin long integer value names with ‘li’	I	C/C++
NamePointerVar	Prefix variable type pointer names with ‘p’	I	C/C++
NameShortInt	Begin short integer variable names with ‘si’	I	C/C++
NameSignedChar	Begin signed character variable names with ‘c’	I	C/C++
NameString	Begin terminated characters’ string variable names with ‘sz’	I	C/C++
NameStructType	Begin struct type names with an uppercase letter that is prefixed by software element and suffixed by ‘_t’	I	C/C++

Item	Description	Severity	C/C++
NameUnsignedChar	Begin unsigned character type names with ‘uc’	I	C/C++
NameUnsignedInt	Begin unsigned integer type variables with ‘ui’	I	C/C++
NameVariable	Begin variable names with a lowercase letter	I	C/C++
NamingStructUnion Members	Use lowercase letters for structure and union member names	PV	C/C++
NonScalarTypedefs	Append names of non-scalar typedefs with "_t"	PV	C/C++
NumberFunctionParam	Avoid functions with more than 5 parameters	V	C/C++
OpEqualThis	Return reference to `*this` in `operator=` functions	SV	C++
PassByValue	Pass built-in types by value unless you are modifying them	V	C/C++
PointerParam Dereference	Don’t dereference possibly null pointer parameters	PV	C
PublicInterface	Avoid member variables in the public interface	I	C++
ReferenceInitialization	Do not initialize a reference to refer to an object whose address can be changed	PV	C++
SingularSwitch Statement	Avoid switch statements with only one case	PV	C/C++
SourceFileSize	Avoid source files that are longer than 500 lines	I	C/C++
SourceNaming Convention	Use the “.c” extension for source file names	I	C/C++
StructKeyword	Do not use a struct keyword to declare a variable in C++	I	C++

Item	Description	Severity	C/C++
ThrowDestructor	Do not throw from within a destructor	V	C++
TooManyFields	Avoid structs, unions, or classes with more than 20 fields	I	C/C++
UnionFieldNotDefined	Define fields for union declarations	I	C/C++
UnnecessaryCast	Avoid unnecessary casts	V	C++
Unnecessary Equal	Avoid unnecessary `==true`s	I	C/C++
UnreachableCode	Do not use unreachable code	PV	C/C++
UnusedLocalVariable	Avoid unused local variables	PV	C/C++
UnusedParameter	Avoid unused parameters	PSV	C/C++
Unused PrivateMember	Avoid unused private member variables	V	C++
UsePositiveLogic	Use positive logic rather than negative logic whenever practical	PV	C/C++