|  |
| --- |
| MET Nightly build  Version 1.0  Code analysis |

|  |
| --- |
| **By: met**  **2023-01-11** |

# Content

[Content 1](#_Toc97156258)

[Introduction 2](#_Toc97156259)

[Configuration 2](#_Toc97156260)

[Synthesis 3](#_Toc97156261)

[Analysis Status 3](#_Toc97156262)

[Quality gate status 3](#_Toc97156263)

[Metrics 3](#_Toc97156264)

[Tests 3](#_Toc97156265)

[Detailed technical debt 3](#_Toc97156266)

[Metrics Range 5](#_Toc97156267)

[Volume 5](#_Toc97156268)

[Issues 6](#_Toc97156269)

[Charts 6](#_Toc97156270)

[Issues count by severity and type 8](#_Toc97156271)

[Issues List 8](#_Toc97156272)

[Security Hotspots 9](#_Toc97156273)

[Security hotspots count by category and priority 9](#_Toc97156274)

[Security hotspots List 9](#_Toc97156275)

# Introduction

This document contains results of the code analysis of MET Nightly build.

# Configuration

* Quality Profiles
  + Names: Sonar way [C]; Sonar way [C++]; Sonar way [Python];
  + Files: AYNrJr7lEG9UUXOvumWd.json; AYNrJr4EEG9UUXOvumJZ.json; AYNrJrpIEG9UUXOvulLt.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

## Analysis Status

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Security Review | Maintainability |
| E.png | **A.png** | **E.png** | **A.png** |

## Quality gate status

|  |  |
| --- | --- |
| Quality Gate Status | **ERROR.png** |

|  |  |
| --- | --- |
| Metric | Value |
| Reliability Rating on New Code | ERROR (C is worse than A) |
| Security Rating on New Code | OK |
| Maintainability Rating on New Code | OK |
| Coverage on New Code | ERROR (0.0% is less than 80%) |
| Duplicated Lines (%) on New Code | ERROR (3.5% is greater than 3%) |
| Security Hotspots Reviewed on New Code | ERROR (0.0% is less than 100%) |

## Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Coverage | Duplication | Comment  density | Median number of lines of code per file | Adherence to coding standard |
| 0.0 % | **4.6 %** | **13.0 %** | **84.0** | **97.1 %** |

## Tests

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Total | Success Rate | Skipped | Errors | Failures |
| 0 | **0 %** | **0** | **0** | **0** |

## Detailed technical debt

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Maintainability | Total |
| 1d 1h 5min | - | 403d 5h 36min | 404d 6h 41min |

## Metrics Range

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| Max | 36041.0 | 37540.0 | 183808.0 | 75.8 | 0.0 | 91.8 |

## Volume

|  |  |
| --- | --- |
| Language | Number |
| C | 25594 |
| C++ | 158202 |
| Python | 12 |
| Total | 183808 |

# Issues

## Charts

## Issues count by severity and type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type / Severity | INFO | MINOR | MAJOR | CRITICAL | BLOCKER |
| BUG | 0 | 0 | 27 | 0 | 11 |
| VULNERABILITY | 0 | 0 | 0 | 0 | 0 |
| CODE\_SMELL | 3 | 1976 | 4428 | 3539 | 16 |

## Issues List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| Memory access should be explicitly bounded to prevent buffer overflows | Array overruns and buffer overflows happen when memory access accidentally goes beyond the boundary of the allocated array or buffer. These overreaching accesses cause some of the most damaging, and hard to track defects. Noncompliant Code Example int array[10]; array[10] = 0; // Noncompliant: index should be between 0 &amp; 9 char \*buffer1 = (char \*) malloc(100); char \*buffer2 = (char \*) malloc(50); memcpy(buffer2, buffer1, 100); // Noncompliant: buffer2 will overflow. Compliant Solution int array[10]; array[9] = 0; char \*buffer1 = (char \*) malloc(100); char \*buffer2 = (char \*) malloc(50); memcpy(buffer2, buffer1, 50); See MITRE, CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer MITRE, CWE-131 - Incorrect Calculation of Buffer Size MITRE, CWE-788 - Access of Memory Location After End of Buffer CERT, ARR30-C. - Do not form or use out-of-bounds pointers or array subscripts CERT, STR50-CPP. - Guarantee that storage for strings has sufficient space for character data and the null terminator | BUG | BLOCKER | 9 |
| "memcpy", "memmove", and "memset" should only be called with pointers to trivially copyable types | The functions memcpy, memmove and memset can only be used for objects of trivially copyable types. This includes scalar types, arrays, and trivially copyable classes. A class type is trivially copyable if: One or more of the following special member functions is trivial and the rest are deleted: copy constructor, move constructor, copy assignment operator, and move assignment operator, It has a trivial, non-deleted destructor, It has trivially copyable members and base classes, It has no virtual functions. Note: a default implementation, both explicit (with =default) or implicit (if the special member function is omitted), is considered trivial. Noncompliant Code Example class Shape { public: int x; int y; virtual ~Shape(); // This makes the class non trivially copyable }; void f(Shape \*dest, Shape \*source) { memcpy(dest, source, sizeof Shape); // Noncompliant } Compliant Solution class Shape { public: int x; int y; virtual ~Shape(); // This makes the class non trivially copyable }; void f(Shape \*dest, Shape \*source) { (\*dest) = (\*source); } | BUG | BLOCKER | 2 |
| All code should be reachable | Some statements (return, break, continue, goto, co\_return) and throw expressions move control flow out of the current code block. Furthermore, some function do not return control flow (e.g. abort(), std::terminate(), functions with the [[noreturn]] attribute). Any unlabeled statements that come after such a jump or function call are unreachable, and either this dead code should be removed, or the logic should be corrected. Noncompliant Code Example int fun(int a) { int i = 10; return i + a; // Noncompliant i++; // dead code } Compliant Solution int fun(int a) { int i = 10; return i + a; } See MISRA C:2004, 14.1 - There shall be no unreachable code MISRA C++:2008, 0-1-1 - A project shall not contain unreachable code MISRA C:2012, 2.1 - A project shall not contain unreachable code MITRE, CWE-561 - Dead Code CERT, MSC56-J. - Detect and remove superfluous code and values CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | BUG | MAJOR | 24 |
| Variables should be initialized before use | Variables should be initialized before their use to avoid unexpected behaviors due to garbage values. Noncompliant Code Example int function(int flag, int b) { int a; if (flag) { a = b; } return a; // Noncompliant - "a" has not been initialized in all paths } Compliant Solution int function(int flag, int b) { int a = 0; if (flag) { a = b; } return a; } See MITRE, CWE-457 - Use of Uninitialized Variable MISRA C:2004, 9.1 - All automatic variables shall have been assigned a value before being used. MISRA C++:2008, 8-5-1 - All variables shall have a defined value before they are used. | BUG | MAJOR | 3 |
| Context-sensitive keywords should not be used as identifiers | The C++ standards define some identifiers as having special meaning in specific contexts. These are: final and override since C++11 module and import since C++20 While it is technically possible to use them as normal identifiers, it’s clearer for the reader of the code to consider them as if they were keywords, and only use them with their special meaning. Noncompliant Code Example void module(int final); // Noncompliant Compliant Solution void precept(int finalValue); // Compliant | CODE\_SMELL | BLOCKER | 2 |
| Non-reentrant POSIX functions should be replaced with their reentrant versions |  | CODE\_SMELL | BLOCKER | 2 |
| Move and swap operations should be "noexcept" |  | CODE\_SMELL | BLOCKER | 6 |
| "goto" should jump to labels declared later in the same function | Unconstrained use of goto can lead to programs that are extremely difficult to comprehend and analyse. For C++, it can also lead to the program exhibiting unspecified behavior. However, in many cases a total ban on goto requires the introduction of flags to ensure correct control flow, and it is possible that these flags may themselves be less transparent than the goto they replace. Therefore, the restricted use of goto is allowed where that use will not lead to semantics contrary to developer expectations. "Back" jumps are prohibited, since they can be used to create iterations without using the well-defined iteration statements supplied by the core language. Noncompliant Code Example int f() { int j = 0; L1: ++j; if (10 == j) { goto L2; // forward jump ignored } // ... goto L1; // Noncompliant L2: return ++j; } Compliant Solution int f() { for (int j = 0; j &lt; 11; j++) { // ... } return ++j; } See MISRA C++:2008, 6-6-2 - The goto statement shall jump to a label declared later in the same function body MISRA C:2012, 15.2 - The goto statement shall jump to a label declared later in the same function | CODE\_SMELL | BLOCKER | 6 |
| Methods should not be empty | There are several reasons for a method not to have a method body: It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production. It is not yet, or never will be, supported. In this case an exception should be thrown in languages where that mechanism is available. The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override. Noncompliant Code Example void fun(int p1) { } Compliant Solution void fun(int p1) { int a = doSomething(p1); int threshold = 42; if (a &gt; threshold) { // ... } } or void fun(int p1) { // Intentionally unimplemented... } Exceptions This rule doesn’t raise an issue for empty class constructors or destructors. For instance this is the only way to define user-defined default constructors. | CODE\_SMELL | CRITICAL | 1 |
| C-style memory allocation routines should not be used |  | CODE\_SMELL | CRITICAL | 13 |
| Inherited functions should not be hidden |  | CODE\_SMELL | CRITICAL | 29 |
| Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply | Nested if, for, do, while, switch and try statements is a key ingredient for making what’s known as "Spaghetti code". Such code is hard to read, refactor and therefore maintain. Noncompliant Code Example With the default threshold of 3: if (condition1) { // Compliant; depth = 1 /\* ... \*/ if (condition2) { // Compliant; depth = 2 /\* ... \*/ for(int i = 0; i &lt; 10; i++) { // Compliant; depth = 3, not exceeding the limit /\* ... \*/ if (condition4) { // Noncompliant; depth = 4 if (condition5) { // Depth = 5, exceeding the limit, but issues are only reported on depth = 4 /\* ... \*/ } return; } } } } Exceptions Each use of a macro containing control flow statements is counted as one nesting level, even if the macro contains more than one control flow statement. #define FOREACH(V,ARR) if(ARR!=nullptr) for(int V=0; V&lt;(sizeof(ARR)/sizeof(ARR[0])); V++) if (condition1) { // Compliant; depth = 1 if (condition2) { // Compliant; depth = 2 FOREACH(i, arr) { // Compliant; depth = 3 (not 4) if (condition3) { // Noncompliant; depth = 4 /\* ... \*/ } } } } | CODE\_SMELL | CRITICAL | 414 |
| Constructors and destructors should only use defined methods and fields | Calling methods or fields which are not initialized in constructors or destructors can lead to undefined behavior. For example: Calling an overridable member function from a constructor or destructor could result in unexpected behavior when instantiating a subclass which overrides the member function. By contract, the subclass class constructor starts by calling the parent class constructor. The parent class constructor calls the parent member function and not the one overridden in the child class, which is confusing for child class' developer. It can produce an undefined behavior if the member function is pure virtual in the parent class. Noncompliant Code Example class Parent { public: Parent() { method1(); method2(); // Noncompliant; confusing because Parent::method2() will always been called even if the method is overridden } Parent(int i):field(i) {} virtual ~Parent() { method3(); // Noncompliant; undefined behavior (ex: throws a "pure virtual method called" exception) } protected: int field; int method1() { /\*...\*/ } virtual void method2() { /\*...\*/ } virtual void method3() = 0; // pure virtual }; class Child : public Parent { public: Child() { // leads to a call to Parent::method2(), not Child::method2() } Child() : Parent(field) {} // Noncompliant; "field" is not initialized yet Child() : Parent(method1()) {} // Noncompliant; "method1" is not initialized yet virtual ~Child() { method3(); // Noncompliant; Child::method3() will always be called even if a child class overrides method3 } protected: void method2() override { /\*...\*/ } void method3() override { /\*...\*/ } }; Compliant Solution class Parent { public: Parent() { method1(); Parent::method2(); // acceptable but poor design } virtual ~Parent() { // call to pure virtual function removed } protected: void method1() { /\*...\*/ } virtual void method2() { /\*...\*/ } virtual void method3() = 0; }; class Child : public Parent { public: Child() { } virtual ~Child() { method3(); // method3() is now final so this is okay } protected: void method2() override { /\*...\*/ } void method3() final { /\*...\*/ } // this virtual function is "final" }; See CERT, MET05-J. - Ensure that constructors do not call overridable methods CERT, OOP50-CPP. - Do not invoke virtual functions from constructors or destructors | CODE\_SMELL | CRITICAL | 4 |
| "explicit" should be used on single-parameter constructors and conversion operators |  | CODE\_SMELL | CRITICAL | 21 |
| Special member function should not be defined unless a non standard behavior is required |  | CODE\_SMELL | CRITICAL | 15 |
| When the "Rule-of-Zero" is not applicable, the "Rule-of-Five" should be followed |  | CODE\_SMELL | CRITICAL | 29 |
| Member variables should not be "protected" | Protected member variables are similar to global variables; any derived class can modify them. When protected member variables are used, invariants cannot be enforced. Also, protected member variables are hard to maintain since they can be manipulated through multiple classes in different files. If a class is just a data store without logic, it can safely contain only public member variables and no member functions. Otherwise, data members are tightly coupled to the class' logic, and encapsulation must be used. In this case, having only private member variables enforces invariants for data and ensures that logic is defined only in the member functions of the class. Structuring it this way makes It easier to guarantee integrity and easier for maintainers to understand the code. But when an object provides encapsulation by using protected member variables, data integrity logic can be spread through the class and all its derived class, becoming a source of complexity and that will be error-prone for maintainers and extenders. That’s why protected member variables should be changed to private and manipulated exclusively through public or protected member functions of the base class. This rule raises an issue when a class or struct contains protected member variables. Noncompliant Code Example class Stat { public: long int getCount() { return count; } protected: long int count = 0; // Noncompliant; expose a protected member variable. // By just looking at "Stat" class, it's not possible to be sure that "count" // is modified properly, we also need to check all derived classes }; class EventStat : public Stat { public: void onEvent() { if (count &lt; LONG\_MAX) { count++; } } }; Compliant Solution class Stat { public: long int getCount() { return count; } protected: void increment() { // Compliant; expose a protected member function if (count &lt; LONG\_MAX) { count++; } } private: long int count = 0; // member variable is private }; class EventStat : public Stat { public: void onEvent() { increment(); } }; Exceptions Const member variables and reference member variables are ignored since they don’t break invariants. See MISRA C++:2008, 11-0-1 - Member data in non-POD class types shall be private. C++ Core Guidelines C.133 - Avoid protected data | CODE\_SMELL | CRITICAL | 216 |
| Cognitive Complexity of functions should not be too high | Cognitive Complexity is a measure of how hard the control flow of a function is to understand. Functions with high Cognitive Complexity will be difficult to maintain. See Cognitive Complexity | CODE\_SMELL | CRITICAL | 235 |
| A conditionally executed single line should be denoted by indentation | In the absence of enclosing curly braces, the line immediately after a conditional is the one that is conditionally executed. By both convention and good practice, such lines are indented. In the absence of both curly braces and indentation the intent of the original programmer is entirely unclear and perhaps not actually what is executed. Additionally, such code is highly likely to be confusing to maintainers. Noncompliant Code Example if (condition) // Noncompliant doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo(); Compliant Solution if (condition) doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo(); | CODE\_SMELL | CRITICAL | 4 |
| "nullptr" should be used to denote the null pointer | Before C++11, the only way to refer to a null pointer was by using the integer literal 0, which created ambiguity with regard to whether a pointer or an integer was intended. Even with the NULL macro, the underlying value is still 0. C++11 introduced the keyword nullptr, which is unambiguous and should be used systematically. Noncompliant Code Example void f(char \*c); void g(int i); void h() { f(0); // Noncompliant f(NULL); // Noncompliant g(0); // Compliant, a real integer g(NULL); // Noncompliant, NULL should not be used for a real integer } Compliant Solution void f(char \*c); void g(int i); void h() { f(nullptr); // Compliant g(0); // Compliant, a real integer } See C++ Core Guidelines ES.47 - Use nullptr rather than 0 or NULL | CODE\_SMELL | CRITICAL | 945 |
| The "Rule-of-Zero" should be followed |  | CODE\_SMELL | CRITICAL | 82 |
| "void \*" should not be used in typedefs, member variables, function parameters or return type | void\* is a pointer to memory of unknown type, and therefore works outside of the safety net provided by the type system. While it can be useful in a function body to interface with external code, there is no good reason to step out of the robust C++ type system when defining a function, either for the function parameters, or for the function return type. For the same reasons, having a member variable of type void\* is not recommended. If you want to work with raw memory buffer, use unsigned char \* (or byte \* if your compiler supports it). If you want to work with different types of data, define a function template and use typed pointers, instead of void \*. If you want a single object to be able to stores objects of different types, std::any can also be a type-safe alternative to void\*. If you want to provide to users of an API an opaque type, declare a type and don’t provide its definition (like with FILE\*). Note that void\* is commonly used to communicate data of unknown type with C code. This rule will nevertheless raise an issue in this case, but it can be ignored. Noncompliant Code Example void saveBuffer(void \*buffer, size\_t size); // Noncompliant void duplicate(void\* destination, size\_t count, void \*source, size\_t size); // Noncompliant class Process { // ... void \*userData; }; using UserData = void\*; // Noncompliant Compliant Solution void saveBuffer(unsigned char \*buffer, size\_t size); template&lt;class T&gt; void duplicate(T\* destination, size\_t count, T \*source); class Process { // ... std::any userData; }; Exceptions void\* can be useful when interfacing with C. As such, the rule will ignore extern "C" functions, as well as types with standard layout. See C++ Core Guidelines I.4 - Make interfaces precisely and strongly typed C++ Core Guidelines T.3 - Use templates to express containers and ranges | CODE\_SMELL | CRITICAL | 11 |
| Memory should not be managed manually |  | CODE\_SMELL | CRITICAL | 770 |
| Macros should not be used to define constants | A macro is a textual replacement, which means that it’s not respecting the type system, it’s not respecting scoping rules…​ There is no reason not to use a constant instead. Most of the time, a macro can be replaced by a constexpr declaration (a constant that is guaranteed to be computed during compilation). If your compiler is too old to properly handle constexpr, you may use const instead. If you have a series of related integer macros, you might also consider replacing them by an enum. Noncompliant Code Example #define MAX\_MEMORY 640 // Noncompliant #define LEFT 0 // Noncompliant #define RIGHT 1 // Noncompliant #define JUMP 2 // Noncompliant #define SHOOT 3 // Noncompliant Compliant Solution constexpr size\_t MAX\_MEMORY = 640; enum class Actions {Left, Right, Jump, Shoot}; See C++ Core Guidelines - ES.31 - Don’t use macros for constants or “functions” C++ Core Guidelines - Enum.1 - Prefer enumerations over macros | CODE\_SMELL | CRITICAL | 123 |
| Argument of "printf" should be a format string | It is a security vulnerability to call printf with a unique string argument which is not a string literal. Indeed, if this argument comes from a user input, this user can : make the program crash, by executing code equivalent to: printf("%s%s%s%s%s%s%s%s") view the stack or a memory at any location, by executing code equivalent to: printf("%08x %08x %08x %08x %08x\n") Noncompliant Code Example void f(char\* userInput) { printf(userInput); // Noncompliant } Compliant Solution void f(char\* userInput) { printf("%s", userInput); // Compliant } See Owasp: format string attack | CODE\_SMELL | CRITICAL | 10 |
| Size of variable length arrays should be positive | Variable length arrays should have a well-defined, positive size. Noncompliant Code Example void f1() { int n; int a[n]; // Noncompliant; n is undefined } void f2() { int n = 0; int a[n]; // Noncompliant; array of zero size } | CODE\_SMELL | CRITICAL | 1 |
| Non-const global variables should not be used | A global variable can be modified from anywhere in the program. At first, this might look convenient, but in fact, it makes programs very hard to understand: When you see a function call, you cannot know if the function will affect the value of the variable or not. You have lost the ability to reason locally about your code and must always have the whole program in mind. Additionally, in multi-threaded environments, global variables are often subject to race conditions. Some global variables defined in external libraries (such as std::cout, std::cin, std::cerr) are fine to use, but you should have a good reason to create your own. If you do use a global variable make sure that they can be safely accessed concurrently. This rule detects all declarations of global variables (in the global namespace or in any namespace) that are not constant. Noncompliant Code Example double oneFoot = 0.3048; double userValue; void readValue(); void writeResult(); int main() { readValue(); writeResult(); } Compliant Solution constexpr double footToMeter = 0.3048; double readValue(); void writeResult(double); int main() { auto userValue = readValue(); writeResult(userValue \* footToMeter); } See C++ Core Guidelines I.2 - Avoid non-const global variables | CODE\_SMELL | CRITICAL | 594 |
| Use discriminated unions or "std::variant" |  | CODE\_SMELL | CRITICAL | 6 |
| A cast shall not remove any const or volatile qualification from the type of a pointer or reference |  | CODE\_SMELL | CRITICAL | 16 |
| Track uses of "TODO" tags | TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example void foo() { // TODO } See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | INFO | 3 |
| Collapsible "if" statements should be merged | Merging collapsible if statements increases the code’s readability. Noncompliant Code Example if (condition1) { if (condition2) { // NonCompliant /\* ... \*/ } } Compliant Solution if (condition1 &amp;&amp; condition2) { // Compliant /\* ... \*/ } | CODE\_SMELL | MAJOR | 130 |
| Unused "private" fields should be removed | If a private field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for. Noncompliant Code Example class MyClass { private: int foo = 42; // Noncompliant, foo is unused public: int compute(int a) { return a \* 42; } }; Compliant Solution class MyClass { public: int compute(int a) { return a \* 42; } }; | CODE\_SMELL | MAJOR | 1 |
| Functions should not have too many parameters | A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the function is doing too many things. Noncompliant Code Example With a maximum number of 4 parameters: void doSomething(int param1, int param2, int param3, int param4, int param5) { ... } Compliant Solution void doSomething(int param1, int param2, int param3, int param4) { ... } See C++ Core Guidelines I.23: Keep the number of function arguments low | CODE\_SMELL | MAJOR | 46 |
| Nested blocks of code should not be left empty | Most of the time a block of code is empty when a piece of code is really missing. So such empty block must be either filled or removed. Noncompliant Code Example void foo() { int x; if (x == 42) /\* Noncompliant - the following nested block is empty \*/ { } else { printf("x != 42"); } } void bar() /\* Compliant - functions are not nested blocks \*/ { } Compliant Solution void foo() { int x; if (x != 42) /\* Compliant \*/ { printf("x != 42"); } } /\* ... \*/ Exceptions When a block contains a comment, this block is not considered to be empty. | CODE\_SMELL | MAJOR | 9 |
| Redundant pairs of parentheses should be removed |  | CODE\_SMELL | MAJOR | 1033 |
| Variables should not be shadowed | Overriding or shadowing a variable declared in an outer scope can strongly impact the readability, and therefore the maintainability, of a piece of code. Further, it could lead maintainers to introduce bugs because they think they’re using one variable but are really using another. Noncompliant Code Example class Foo { public: void doSomething(); private: int myField; }; void Foo::doSomething() { int myField = 0; // Noncompliant // ... } void f(int x, bool b) { int y = 4; if (b) { int x = 7; // Noncompliant int y = 9; // Noncompliant // ... } } Compliant Solution class Foo { public: void doSomething(); private: int myField; }; void Foo::doSomething() { int myInternalField = 0; // Compliant // ... } void f(int x, bool b) { int y = 4; if (b) { int z = 7; // Better yet: Use meaningful names int w = 9; // ... } } Exceptions It is common in a constructor to have constructor arguments shadowing the fields that they will initialize. This pattern avoids the need to select new names for the constructor arguments, and will not be reported by this rule: class Point{ public: Point(int x, int y) : x(x), y(y) {} // Compliant by exception private: int x; int y; }; See MISRA C:2004, 5.2 - Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier MISRA C++:2008, 2-10-2 - Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope MISRA C:2012, 5.3 - An identifier declared in an inner scope shall not hide an identifier declared in an outer scope CERT, DCL01-C. - Do not reuse variable names in subscopes CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes | CODE\_SMELL | MAJOR | 49 |
| Assignments should not be made from within sub-expressions | Assignments within sub-expressions are hard to spot and therefore make the code less readable. Ideally, sub-expressions should not have side-effects. Noncompliant Code Example if ((str = cont.substring(pos1, pos2)).isEmpty()) { // Noncompliant //... Compliant Solution str = cont.substring(pos1, pos2); if (str.isEmpty()) { //... Exceptions Assignments explicitly enclosed in parentheses are ignored. while ((run = keepRunning())) { //... } See MISRA C:2004, 13.1 - Assignment operators shall not be used in expressions that yield a Boolean value MISRA C++:2008, 6-2-1 - Assignment operators shall not be used in sub-expressions MISRA C:2012, 13.4 - The result of an assignment operator should not be used MITRE, CWE-481 - Assigning instead of Comparing CERT, EXP45-C. - Do not perform assignments in selection statements CERT, EXP51-J. - Do not perform assignments in conditional expressions | CODE\_SMELL | MAJOR | 439 |
| Track uses of "FIXME" tags | FIXME tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example int divide(int numerator, int denominator) { return numerator / denominator; // FIXME denominator value might be 0 } See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | MAJOR | 1 |
| Unused functions and methods should be removed | A function that is never called is dead code, and should be removed. Cleaning out dead code decreases the size of the maintained codebase, making it easier to understand the program and preventing bugs from being introduced. This rule detects functions that are never called from inside a translation unit, and cannot be called from the outside. Noncompliant Code Example static void unusedStaticFunction() { // Noncompliant } class Server { public: void start() { // Compliant, the member function "start()" is public log("start"); } private: void clear() { // Noncompliant, the member function "clear()" is unused } void log(const char \* msg) { // Compliant, the member function "log()" is used in "start() { ... }" printf(msg); } }; See MISRA C++:2008, 0-1-10 - Every defined function shall be called at least once. | CODE\_SMELL | MAJOR | 3 |
| Unused function parameters should be removed | Unused parameters are misleading. Whatever the values passed to such parameters, the behavior will be the same. There are some cases when you want to have an unused parameter (usually because the function has to conform to a fixed prototype, because it is virtual or it is going to be called from a template). In this case, and if the parameter is never used, an accepted practice is to leave it unnamed. If it is only sometimes used (for instance, depending on conditional compilation), you may, since C++17, use the [[maybe\_unused]] attribute to be explicit about it. void f([[maybe\_unused]] int i) { assert(i &lt; 42); // In optimized mode, this assert will be removed, and i will be unused } In case of Objective-C it is acceptable to have unused parameters if the method is supposed to be overridden. Noncompliant Code Example void doSomething(int a, int b) { // Noncompliant, "b" is unused compute(a); } Compliant Solution void doSomething(int a) { compute(a); } See MISRA C++:2008, 0-1-11 - There shall be no unused parameters (named or unnamed) in nonvirtual functions. MISRA C:2012, 2.7 - There should be no unused parameters in functions CERT, MSC12-C. - Detect and remove code that has no effect or is never executed C++ Core Guidelines - F.9 - Unused parameters should be unnamed | CODE\_SMELL | MAJOR | 35 |
| Polymorphic base class destructor should be either public virtual or protected non-virtual | When a class with no virtual destructor is used as a base class, surprises can occur if pointers to instances of this class are used. Specifically, if an instance of a derived class is deleted through a pointer to the base type, the behavior is undefined and can lead to resource leaks, crashes or corrupted memory. If it is not expected for base class pointers to be deleted, then the destructor should be made protected to avoid such a misuse. Noncompliant Code Example class Base { // Noncompliant: no destructor is supplied, and the default version is not virtual public: Base() {} virtual void doSomething() {} }; class Derived : public Base { } void f() { Base \*p = new Derived(); delete p; // Undefined behavior } Compliant Solution class Base { public: Base() {} virtual ~Base() = default; virtual void doSomething() {} }; See CERT, OOP52-CPP. - Do not delete a polymorphic object without a virtual destructor Virtuality article C++ Core Guidelines C.35 - A base class destructor should be either public and virtual, or protected and nonvirtual C++ Core Guidelines C.127 - A class with a virtual function should have a virtual or protected destructor | CODE\_SMELL | MAJOR | 5 |
| Pass by reference to const should be used for large input parameters |  | CODE\_SMELL | MAJOR | 152 |
| Sections of code should not be commented out | Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required. Exceptions This rule does not apply to code documentation using Doxygen, QDoc, markdown, or HTML tags. See MISRA C:2004, 2.4 - Sections of code should not be "commented out". MISRA C++:2008, 2-7-2 - Sections of code shall not be "commented out" using C-style comments. MISRA C++:2008, 2-7-3 - Sections of code should not be "commented out" using C++ comments. MISRA C:2012, Dir. 4.4 - Sections of code should not be "commented out" | CODE\_SMELL | MAJOR | 243 |
| "switch" statements should not have too many "case" clauses | When switch statements have large sets of case clauses, it is usually an attempt to map two sets of data. A real map structure would be more readable and maintainable, and should be used instead. | CODE\_SMELL | MAJOR | 3 |
| Structures should not have too many fields | A structure, such as a struct, union or class that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain, and having a lot of fields is an indication that a structure has grown too large. Above a specific threshold, it is strongly advised to refactor the structure into smaller ones that focus on well defined topics. | CODE\_SMELL | MAJOR | 17 |
| Unused assignments should be removed | A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction. Calculating or retrieving a value only to then overwrite it or throw it away, could indicate a serious error in the code. Even if it’s not an error, it is at best a waste of resources. Therefore all calculated values should be used. Noncompliant Code Example i = a + b; // Noncompliant; calculation result not used before value is overwritten i = compute(); Compliant Solution i = a + b; i += compute(); Exceptions This rule ignores: variable declarations initializers prefix and postfix increments and decrements x++; null pointer assignments x = NULL; self assignments (i.e. x = x;) See MITRE, CWE-563 - Assignment to Variable without Use ('Unused Variable') CERT, MSC13-C. - Detect and remove unused values CERT, MSC56-J. - Detect and remove superfluous code and values | CODE\_SMELL | MAJOR | 44 |
| Two branches in a conditional structure should not have exactly the same implementation |  | CODE\_SMELL | MAJOR | 13 |
| "static" members should be accessed statically | While it is possible to access static members from a class instance, it’s bad form, and considered by most to be misleading because it implies to the readers of your code that there’s an instance of the member per class instance. Noncompliant Code Example class MyClass { public : static void Mymethod() { // ... } }; MyClass\* pmyclass = new MyClass(); pmyclass-&gt;Mymethod(); // Noncompliant Compliant Solution class MyClass { public : static Mymethod() { // ... } }; Myclass::Mymethod(); | CODE\_SMELL | MAJOR | 49 |
| Multiline blocks should be enclosed in curly braces | Curly braces can be omitted from a one-line block, such as with an if statement or for loop, but doing so can be misleading and induce bugs. This rule raises an issue when the whitespacing of the lines after a one line block indicates an intent to include those lines in the block, but the omission of curly braces means the lines will be unconditionally executed once. Note that this rule considers tab characters to be equivalent to 1 space. If you mix spaces and tabs you will sometimes see issues in code which look fine in your editor but are confusing when you change the size of tabs. Noncompliant Code Example if (condition) firstActionInBlock(); secondAction(); // Noncompliant; executed unconditionally thirdAction(); if (condition) firstActionInBlock(); secondAction(); // Noncompliant; secondAction executed unconditionally if (condition) firstActionInBlock(); // Noncompliant secondAction(); // Executed unconditionally if (condition); secondAction(); // Noncompliant; secondAction executed unconditionally String str = null; for (int i = 0; i &lt; array.length; i++) str = array[i]; doTheThing(str); // Noncompliant; executed only on last array element Compliant Solution if (condition) { firstActionInBlock(); secondAction(); } thirdAction(); String str = null; for (int i = 0; i &lt; array.length; i++) { str = array[i]; doTheThing(str); } See MITRE, CWE-483 - Incorrect Block Delimitation | CODE\_SMELL | MAJOR | 1 |
| Member data should be initialized in-class or in a constructor initialization list | There are three ways to initialize a non-static data member in a class: With an in-class initializer (since C++11) In the initialization list of a constructor In the constructor body You should use those methods in that order of preference. When applicable, in-class initializers are best, because they apply automatically to all constructors of the class (except for default copy/move constructors and constructors where an explicit initialization for this member is provided). But they can only be used for initialization with constant values. If your member value depends on a parameter, you can initialize it in the constructor’s initialization list. If the initialization is complex, you can define a function to compute the value, and use this function in the initializer list. Initialization in the constructor body has several issues. First, it’s not an initialization, but an assignment. Which means it will not work with all data types (const-qualified members, members of reference type, member of a type without default constructor…​). And even if it works, the member will first be initialized, then assigned to, which means useless operations will take place. To prevent "use-before-set" errors, it’s better to immediately initialize the member with its real value. It’s hard to find a good example where setting the value of a member in the constructor would be appropriate. One case might be when you assign to several data members in one operation. As a consequence constructor bodies are empty in many situations. This rules raises an issue in two conditions: When you assign a value to a member variable in the body of a constructor. When you default-initialize in an initializer list a member variable, that would be value-initialized by default For C++11 or later, when you initialize a member variable in the initializer list of a constructor, but could have done so directly in the class: The variable has either no in-class initializer, or an in-class initializer with the same value as in the constructor The initial value does not depend on a constructor parameter Noncompliant Code Example class S { int i1; int i2; int i3; public: S( int halfValue, int i2 = 0) : i2(i2), i3(42) { // Noncompliant for i1 and i3, compliant for i2 this-&gt;i1 = 2\*halfValue; } }; Compliant Solution class S { int i1; int i2; int i3 = 42; // In-class initializer public: S( int halfValue, int i2 = 0 ) : i1(2\*halfValue), i2(i2) {} // Compliant }; See C++ Core Guidelines C.48 - Prefer in-class initializers to member initializers in constructors for constant initializers C++ Core Guidelines C.49 - Prefer initialization to assignment in constructors | CODE\_SMELL | MAJOR | 80 |
| Conditional operators should not be nested |  | CODE\_SMELL | MAJOR | 9 |
| Function pointers should not be used as function parameters |  | CODE\_SMELL | MAJOR | 7 |
| Implicit casts should not lower precision | Implicit casts discard information when the resulting type has a lower precision than the original type. Noncompliant Code Example int a = 2.1f; // Noncompliant long double f(); double d = 0; d += f(); // Noncompliant | CODE\_SMELL | MAJOR | 498 |
| Size argument of memory functions should be consistent |  | CODE\_SMELL | MAJOR | 1 |
| Classes should not contain both public and private data members | Mixing (non-const) public and private data members is a bad practice because it causes confusion about the intention of the class: If the class is a collection of loosely related values, all the data members should be public. On the other hand, if the class is trying to maintain an invariant, all the data members should be private. If we mix data members with different levels of accessibility, the purpose of the class is muddled. Noncompliant Code Example class MyClass // Noncompliant { public: int firstNumber1() const { return firstNumber; } void setFirstNumber(int firstNumber) { this-&gt;firstNumber = firstNumber; } int secondNumber = 2; const int constNumber = 0; // const data members are fine private: int firstNumber = 1; }; Compliant Solution class MyClass // Depending on the case, the solution might be different. Here, since this class does not enforce any invariant, we make all the data members public { public: int firstNumber; int secondNumber; const int constNumber = 0; }; Exceptions Since const data members cannot be modified, it’s not breaking encapsulation to make a const value public, even in a class that enforces an invariant. See C++ Core Guidelines C.134: Ensure all non-const data members have the same access level C++ Core Guidelines C.9: Minimize exposure of members | CODE\_SMELL | MAJOR | 19 |
| Member functions that don't mutate their objects should be declared "const" | No member function can be invoked on a const-qualified object unless the member function is declared "const". Qualifying member functions that don’t mutate their object with the "const" qualifier makes your interface easier to understand; you can deduce without diving into implementation if a member function is going to mutate its object. Also, const-qualified member functions make working with const-qualified objects possible. The compiler ensures that only member functions that are declared "const" can be invoked on "const" objects. Avoiding declaring non-mutating member functions const might break&nbsp;const-correctness: it will not be possible to invoke such non-mutating functions on const-qualified objects. Noncompliant Code Example class A { void f(){ // Noncompliant std::cout&lt;&lt; "f doesn't mutate A"; } }; Compliant Solution class A { void f() const { std::cout&lt;&lt; "f doesn't mutate A"; } }; Exceptions Virtual member functions that don’t mutate their objects don’t necessarily need to be declared const. This might be done in order to allow them to be overridden by non-const functions. | CODE\_SMELL | MAJOR | 62 |
| Integer literals should not be cast to bool | Even though C++ provides "true" and "false" as boolean literals, it allows using integer literals in places where boolean type is expected. This can be done through implicit or explicit casting. In contexts where boolean type is expected, integral literals should be avoided. Using boolean literals instead would make your code more readable and less error-prone. Noncompliant Code Example void f(){ bool isX = 1; // Noncompliant bool isY = 0; // Noncompliant bool ternaryIsX = isX ? 1 : isY; // Noncompliant bool cCast= (bool)0; // Noncompliant bool cppCast= static\_cast&lt;bool&gt;(1); // Noncompliant if(1) { // Noncompliant ... } } Compliant Solution void f(){ bool isX = true; bool isY = false; bool ternaryIsX = isX ? true : isY; bool cCast= false; bool cppCast= true; if(true) { ... } } | CODE\_SMELL | MAJOR | 40 |
| "auto" should be used to avoid repetition of types |  | CODE\_SMELL | MAJOR | 316 |
| C-style array should not be used | C-style arrays (such as int i[10]) are not very convenient to use: They are fixed size (even C VLA are not truly variable size, and they are not supported in C++) If the number of elements in the array can vary, it will lead to manual memory allocation (or people will use fixed-size arrays that "should be large enough", which is both a waste of memory and a limitation of the program) It is very easy to lose the size of the array since an array passed to a function decays into a pointer The C++ standard library proposes two types that are better than C-style arrays and together cover all the use cases of C-style arrays: For fixed-size arrays, where the memory is on the stack, use std::array. It is like a C-style array, except that it has a normal argument passing semantic, and the size is always a part of the type. If std::array is not available to you (before C++11), you can roll your own version. For variable-size arrays, use std::vector. It can be resized and handles memory allocation transparently. For character strings, you should use std::string instead of arrays of characters. For arrays of characters that are not strings (e.g., alphabet, exit codes, keyboard control list) perfer std::array or std::vector as per the first two bullets. The rule S945 is related to this rule but focuses on passing arguments of an array type. S5025 will flag the use of dynamic memory allocation that could be replaced by std::vector. Noncompliant Code Example void f() { int a[10]; // Noncompliant } Compliant Solution void f() { std::array&lt;int, 10&gt; a1; // If the size really is a constant // Or std::vector&lt;int&gt;a2; // For variable size auto s = "Hello!"; // Compliant by exception } Exceptions This rule will not report the use of C-style arrays in extern "C" code (since those arrays are often required here for compatibility with external code) and in the arguments of main. See C++ Core Guidelines SL.con.1 - Prefer using STL array or vector instead of a C array | CODE\_SMELL | MAJOR | 365 |
| Emplacement should be prefered when insertion creates a temporary with sequence containers |  | CODE\_SMELL | MAJOR | 20 |
| Transparent comparator should be used with associative "std::string" containers |  | CODE\_SMELL | MAJOR | 71 |
| If a function has internal linkage then all re-declarations shall include the static storage class specifer | This rule is a strict implementation of a MISRA (Motor Industry Software Reliability Association) rule. MISRA defines best practices for developing safety-critical software. You can learn more about this rule in the MISRA documents referenced below. See MISRA C++ 2008, 3-3-2 MISRA C 2004, 8.11 MISRA C 2012, 8.8 | CODE\_SMELL | MAJOR | 499 |
| "bool" expressions should not be used as operands to built-in operators other than =, &&, ||, !, ==, !=, unary &, and the conditional operator |  | CODE\_SMELL | MAJOR | 17 |
| Loops should not have more than one "break" or "goto" statement | Restricting the number of exits from a loop is done in the interests of good structured programming. One break or goto statement is acceptable in a loop since this allows, for example, for dual-outcome loops or optimal coding. Noncompliant Code Example With the default threshold of 1: for (int i = 0; i &lt; 10; i++) { if (...) { break; // Compliant } else if (...) { break; // Non-compliant - second jump from loop } else { ... } } while (...) { if (...) { break; // Compliant } if (...) { break; // Non-compliant - second jump from loop } } Compliant Solution for (int i = 0; i &lt; 10; i++) { if (...) { break; // Compliant } } while (...) { if (...) { break; // Compliant } } See MISRA C:2004, 14.6 - For any iteration statement there shall be at most one break statement used for loop termination. MISRA C++:2008, 6-6-4 - For any iteration statement there shall be no more than one break or goto statement used for loop termination. MISRA C:2012, 15.4 - There should be no more than one break or goto statement used to terminate any iteration statement | CODE\_SMELL | MAJOR | 20 |
| #include directives in a file should only be preceded by other preprocessor directives or comments |  | CODE\_SMELL | MAJOR | 131 |
| "/\*" and "//" should not be used within comments | Defining a nested single-line comment within a multi-line comment invites errors. It may lead a developer to wrongly think that the lines located after the single-line comment are not part of the comment. If a comment starting sequence, /\* or //, occurs within a /\* comment, is it quite likely to be caused by a missing \*/ comment ending sequence. If a comment starting sequence occurs within a // comment, it is probably because a region of code has been commented-out using //. Noncompliant Code Example /\* some comment, end comment marker accidentally omitted // Make sure this function is called in a thread safe context Perform\_Critical\_Safety\_Function(X); ... /\* this comment is non-compliant \*/ Exceptions The sequence // is permitted within a // comment. See CERT, MSC04-C. - Use comments consistently and in a readable fashion MISRA C:2004, 2.3 - The character sequence /\* shall not be used within a comment. MISRA C++:2008, 2-7-1 - The character sequence /\* shall not be used within a C-style comment. MISRA C:2012, 3.1 - The character sequences /\* and // shall not be used within a comment | CODE\_SMELL | MINOR | 12 |
| Empty statements should be removed | Empty statements, i.e. ;, are usually introduced by mistake, for example because: It was meant to be replaced by an actual statement, but this was forgotten. There was a typo which lead the semicolon to be doubled, i.e. ;;. Noncompliant Code Example void doSomething() { ; // Noncompliant - was used as a kind of TODO marker } Compliant Solution void doSomething() { } Exceptions In the case of empty expanded macro and in the case of 2 consecutive semi-colons when one of the two is part of a macro-definition then the issue is not raised. Example: #define A(x) x; #define LOG(x) void fun() { A(5); LOG(X); } See MISRA C:2004, 14.3 - Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character. MISRA C++:2008, 6-2-3 - Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character. CERT, MSC12-C. - Detect and remove code that has no effect or is never executed CERT, MSC51-J. - Do not place a semicolon immediately following an if, for, or while condition CERT, EXP15-C. - Do not place a semicolon on the same line as an if, for, or while statement | CODE\_SMELL | MINOR | 10 |
| Do not check emptiness with a size method when a dedicated function exists |  | CODE\_SMELL | MINOR | 19 |
| Overriding member functions should do more than simply call the same member in the base class | Overriding a function just to call the overridden function from the base class without performing any other actions can be useless and misleading. There are cases when it is justified, because redeclaring the function allows some side effects: Changing the visibility of the function in the derived class Preventing the base class function from being hidden by an overload added in the derived class (a using declaration could have the same effect) To resolve ambiguities in cases of multiple inheritance To make an inherited function final This rule raises an issue when an override which is not in one of the aforementioned situation only calls the overridden function, directly forwarding its arguments. Noncompliant Code Example class Base { public: virtual void f(); }; class Derived : public Base { public: virtual void f() { Base::f(); // Noncompliant } }; Compliant Solution class Base { public: virtual void f(); }; class Derived : public Base { }; or class Base { public: void f(); }; class Derived : public Base { private: // change of visibility virtual void f() { Base::f(); } }; or class Base { public: void f(); }; class Derived : public Base { public: void f(int i); void f() { // Prevents hiding by f(int) Base::f(); } }; or class Base { public: virtual void f(); }; class Derived : public Base { public: void f() final { // final Base::f(); } }; | CODE\_SMELL | MINOR | 3 |
| Nested code blocks should not be used | Nested code blocks can be used to create a new scope: variables declared within that block cannot be accessed from the outside, and their lifetime end at the end of the block. While this might seem convenient, using this feature in a function often indicates that it has too many responsibilities and should be refactored into smaller functions. A nested code block is acceptable when it surrounds all the statements inside an alternative of a switch (a case xxx: or a default:) because it prevents variable declarations from polluting other cases. Noncompliant Code Example void f(Cache &amp;c, int data) { int value; { // Noncompliant std::scoped\_lock l(c.getMutex()); if (c.hasKey(data)) { value = c.get(data); } else { value = compute(data); c.set(data, value); } } // Releases the mutex switch(value) { case 1: { // Noncompliant, some statements are outside of the block int result = compute(value); save(result); } log(); break; case 2: // ... } } Compliant Solution int getValue(Cache &amp;c, int data) { std::scoped\_lock l(c.getMutex()); if (c.hasKey(data)) { return c.get(data); } else { value = compute(data); c.set(data, value); return value; } } void f(Cache &amp;c, int data) { int value = getValue(c, data); switch(value) { case 1: { // Compliant, limits the scope of "result" int result = compute(value); save(result); log(); } break; case 2: // ... } | CODE\_SMELL | MINOR | 6 |
| Unused local variables should be removed | If a local variable is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable is used for. Noncompliant Code Example int numberOfMinutes(int hours) { int seconds = 0; // Noncompliant, never used return hours \* 60; } Compliant Solution int numberOfMinutes(int hours) { return hours \* 60; } Exceptions No issue is raised on local variables having the attribute "unused" and on object declarations with non-empty arguments. | CODE\_SMELL | MINOR | 77 |
| Init-declarator-lists and member-declarator-lists should consist of single init-declarators and member-declarators respectively | Where multiple declarators appear in the same declaration the type of an identifier may not meet developer expectations. Noncompliant Code Example int i1; int j1; // Compliant, but not preferred int i2, \*j2; // Noncompliant int \*i3, &amp;j3 = i2; // Noncompliant Compliant Solution int i1; int j1; int i2; int \*j2; int \*i3; int &amp;j3 = i2; See MISRA C++:2008, 8-0-1 - An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively CERT, DCL52-J. - Do not declare more than one variable per declaration CERT, DCL04-C. - Do not declare more than one variable per declaration C++ Core Guidelines - ES.10 - Declare one name (only) per declaration | CODE\_SMELL | MINOR | 764 |
| Redundant casts should not be used | Unnecessary casting expressions make the code harder to read and understand. Noncompliant Code Example int example(int i) { int result = static\_cast&lt;int&gt;(i + 42); // Noncompliant return (int) result; // Noncompliant } Compliant Solution int example(int i) { int result = i + 42; return result; } Exceptions In some rare cases, redundant cast might be justifiable. For example, when casting from platform dependant type; your cast might be redundant on one platform but not on the others. | CODE\_SMELL | MINOR | 71 |
| Members should be initialized in the order they are declared |  | CODE\_SMELL | MINOR | 1 |
| Empty "case" clauses that fall through to the "default" should be omitted | Empty case clauses that fall through to the default are useless. Whether or not such a case is present, the default clause will be invoked. Such cases simply clutter the code, and should be removed. Noncompliant Code Example switch(ch) { case 'a' : handleA(); break; case 'b' : handleB(); break; case 'c' : // Noncompliant default: handleTheRest(); break; } Compliant Solution switch(ch) { case 'a' : handleA(); break; case 'b' : handleB(); break; default: handleTheRest(); break; } | CODE\_SMELL | MINOR | 1 |
| "override" or "final" should be used instead of "virtual" | In a base class, virtual indicates that a function can be overridden. In a derived class, it indicates an override. But given the specifier’s dual meaning, it would be both clearer and more sound to use derived class-specific specifiers instead: override or final. final indicates a function override that cannot itself be overridden. The compiler will issue a warning if the signature does not match the signature of a base-class virtual function. override indicates that a function is intended to override a base-class function. The compiler will issue a warning if this is not the case. It is redundant in combination with final. Noncompliant Code Example class Counter { protected: int c = 0; public: virtual void count() { c++; } }; class FastCounter: public Counter { public: virtual void count() { // Noncompliant c += 2; } }; Compliant Solution class Counter { protected: int c = 0; public: virtual void count() { c++; } }; class FastCounter: public Counter { public: void count() override { c += 2; } }; or class Counter { protected: int c = 0; public: virtual void count() { c++; } }; class FastCounter: public Counter { public: void count() final { c += 2; } }; See S1016 C++ Core Guidelines C.128 - Virtual functions should specify exactly one of virtual, override, or final | CODE\_SMELL | MINOR | 148 |
| The "register" storage class specifier should not be used | According to ISO/IEC 14882 (third edition 2011-09-01) : C++11 The register specifier shall be applied only to names of variables declared in a block or to function parameters. It specifies that the named variable has automatic storage duration. A variable declared without a storage-class-specifier at block scope or declared as a function parameter has automatic storage duration by default. A register specifier is a hint to the implementation that the variable so declared will be heavily used.[ Note: The hint can be ignored and in most implementations it will be ignored if the address of the variable is taken. This use is deprecated - end note ] In upcoming versions of C/C++ we can expect this deprecated specifier to not be supported anymore. | CODE\_SMELL | MINOR | 1 |
| Access specifiers should not be redundant | Redundant access specifiers should be removed because they needlessly clutter the code. Noncompliant Code Example struct S { public: // Noncompliant; does not affect any declaration private: void method(); private: // Noncompliant; does not change accessibility level int member; private: // Noncompliant; does not affect any declaration }; class C { int member; private: // Noncompliant; does not change accessibility level void method(); }; Compliant Solution struct S { private: void method(); int member; }; class C { int member; void method(); }; Exceptions An access specifier at the very beginning of a class or struct that matches the default access level is ignored even when it doesn’t change any accessibility levels. class C { private: // redundant but accepted // ... }; struct S { public: // redundant but accepted // ... }; Such a specifier is redundant, but ignored to allow classes and structs to be described uniformly. class C { public: void call(); protected: int delete(); private: int code; }; struct S { public: // redundant but accepted int sum(); protected: int min(); private: int count; }; | CODE\_SMELL | MINOR | 4 |
| Scoped enumerations should be used | There are two kinds of enumeration: The unscoped enum inherited from C The scoped enumeration enum class or enum struct added in C++ 11 Unscoped enumerations have two major drawbacks that are fixed by scoped enumerations: enum elements are visible from their enclosing scope, instead of requiring the scope resolution operator (ex: Red instead of Color::Red) enum elements convert implicitly to int, so that heterogeneous comparisons such as Red == Big don’t result in compile errors. This rule raises an issue when an unscoped enumeration is used. Noncompliant Code Example enum Color { // Noncompliant; replace this "enum" with "enum class". Red = 0xff0000, Green = 0x00ff00, Blue = 0x0000ff }; enum ProductType { // Noncompliant; replace this "enum" with "enum class". Small = 1, Big = 2 }; void printColor(int color); void printInt(int value); void report() { printColor(Red); // correct printColor(Big); // clearly buggy printInt(Red); // conversion is implicit } Compliant Solution enum class Color { // declared using "enum class" Red = 0xff0000, Green = 0x00ff00, Blue = 0x0000ff }; enum class ProductType { // declared using "enum class" Small = 1, Big = 2 }; void printColor(Color color); // requires "Color" instead of "int" void printInt(int value); void report() { printColor(Color::Red); // correct // printColor(ProductType::Big); =&gt; Compilation error, no known conversion from 'ProductType' to 'Color' printInt(static\_cast&lt;int&gt;(Color::Red)); // conversion never occurs implicitly and must be explicit } Exceptions When the enum is a private member of a class, its use is encapsulated by the class and the drawbacks of unscoped enums can be avoided. Therefore, no issue will be raised in that case. See C++ Core Guidelines Enum.3 - Prefer class enums over “plain” enums | CODE\_SMELL | MINOR | 11 |
| Functions which do not return should be declared as "noreturn" | The attribute noreturn indicates that a function does not return. This information clarifies the behavior of the function and it allows the compiler to do optimizations. It can also help the compiler (and static analyzer tools, i.e. us) provide better error messages: \_\_attribute\_\_((noreturn)) void f(); int g(int b) { if (b == 5) { f(); printf("Hello world\n"); // This is dead code, the compiler/static analyzer can now detect it // There is no returned value, but it is fine, the compiler/static analyzer knows not to warn about it } else { return 3; } } This rule detects when the attribute noreturn can be added to a function. Noncompliant Code Example void g() {&nbsp;// Noncompliant &nbsp;&nbsp;abort(); } Compliant Solution \_\_attribute\_\_((noreturn)) void g() {&nbsp;// or [[noreturn]] for C++ abort(); // Compliant } | CODE\_SMELL | MINOR | 17 |
| Pointer and reference local variables should be "const" if the corresponding object is not modified |  | CODE\_SMELL | MINOR | 82 |
| "using" should be preferred for type aliasing | Since C++11, type aliases can be declared via using or typedef. using should be preferred as more readable because you see the new name/alias first. In addition, using can be templated, which makes it applicable to more situations than typedef. Noncompliant Code Example typedef void (\*FunctionPointerType)(int); Compliant Solution using FunctionPointerType = void (\*)(int); See C++ Core Guidelines - T.43 - Prefer using over typedef for defining aliases | CODE\_SMELL | MINOR | 20 |
| STL algorithms and range-based for loops should be preferred to traditional for loops |  | CODE\_SMELL | MINOR | 72 |
| Return type of functions shouldn't be const qualified value | There is no reason to add a const qualifier to the return type of a function that returns by value. At best, it will be superfluous. At worst, it will be a performance bug: it can prevent move operation (when copy elision doesn’t take place). When an object is const-qualified the copy constructor/assignment will be a better match than the move constructor/assignment. One might think about adding this qualifier in order to forbid the call of unintended functions on the returned object. A common example is to avoid unintended assignments: X x1, x2, x3; if (x1 + x2 = x3) { // Compiler will complain since const object cannot be assigned. Should be "x1 + x2 == x3" ... } C++11 introduced reference qualifiers for member functions. This feature provides a better approach to forbid calling unintended functions: struct X { X&amp; operator=(const X&amp; other) &amp;; }; ... X x1, x2, x3; if (x1 + x2 = x3) { // Compiler will complain since assignment cannot be called on r-value. Should be "x1 + x2 == x3" ... } Noncompliant Code Example class A {...}; const A f(); // Noncompliant Compliant Solution class A {...}; A f(); See C++ Core Guidelines F.20 - Flag returning a const value. | CODE\_SMELL | MINOR | 55 |
| Loop variables should be declared in the minimal possible scope |  | CODE\_SMELL | MINOR | 352 |
| "auto" should be used to store a result of functions that conventionally return an iterator or a range |  | CODE\_SMELL | MINOR | 44 |
| The three expressions of a "for" statement should only be concerned with loop control |  | CODE\_SMELL | MINOR | 21 |
| Pointer and reference parameters should be "const" if the corresponding object is not modified | This rule leads to greater precision in the definition of the function interface. The const qualification shall be applied to the object pointed to, not to the pointer, since it is the object itself that is being protected. Noncompliant Code Example void myfunc ( int \* param1, // object is modified const int \* param2, int \* param3, // Noncompliant int \* param4) // Noncompliant { \*param1 = \*param2 + \*param3 + \*param4; } int main (int argc, const char \* \* argv) // Noncompliant { return argc; } Compliant Solution void myfunc ( int \* param1, // object is modified const int \* param2, const int \* param3, const int \* param4) { \*param1 = \*param2 + \*param3 + \*param4; } int main (int argc, const char \* const \* argv) { return argc; } See MISRA C:2004, 16.7 - A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object. MISRA C++:2008, 7-1-2 - A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified. MISRA C:2012, 8.13 - A pointer should point to a const-qualified type whenever possible | CODE\_SMELL | MINOR | 185 |

# Security Hotspots

## Security hotspots count by category and priority

|  |  |  |  |
| --- | --- | --- | --- |
| Category / Priority | LOW | MEDIUM | HIGH |
| LDAP Injection | 0 | 0 | 0 |
| Object Injection | 0 | 0 | 0 |
| Server-Side Request Forgery (SSRF) | 0 | 0 | 0 |
| XML External Entity (XXE) | 0 | 0 | 0 |
| Insecure Configuration | 0 | 0 | 0 |
| XPath Injection | 0 | 0 | 0 |
| Authentication | 0 | 0 | 0 |
| Weak Cryptography | 0 | 0 | 0 |
| Denial of Service (DoS) | 0 | 0 | 0 |
| Log Injection | 0 | 0 | 0 |
| Cross-Site Request Forgery (CSRF) | 0 | 0 | 0 |
| Open Redirect | 0 | 0 | 0 |
| Permission | 0 | 2 | 0 |
| SQL Injection | 0 | 0 | 0 |
| Encryption of Sensitive Data | 0 | 0 | 0 |
| Traceability | 0 | 0 | 0 |
| Buffer Overflow | 0 | 0 | 21 |
| File Manipulation | 0 | 0 | 0 |
| Code Injection (RCE) | 0 | 0 | 0 |
| Cross-Site Scripting (XSS) | 0 | 0 | 0 |
| Command Injection | 0 | 0 | 0 |
| Path Traversal Injection | 0 | 0 | 0 |
| HTTP Response Splitting | 0 | 0 | 0 |
| Others | 18 | 0 | 0 |

## Security hotspots List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Name | Priority | Severity | Count |
| Others | Using publicly writable directories is security-sensitive | LOW | CRITICAL | 18 |
| Permission | Setting loose POSIX file permissions is security-sensitive | MEDIUM | MAJOR | 2 |
| Buffer Overflow | Using "strcpy" or "wcscpy" is security-sensitive | HIGH | MAJOR | 7 |
| Buffer Overflow | Using "strlen" or "wcslen" is security-sensitive | HIGH | MAJOR | 13 |
| Buffer Overflow | Using "strncpy" or "wcsncpy" is security-sensitive | HIGH | MAJOR | 1 |