Philips Healthcare - C++ Coding Standard

PHILIPS

Version head

issued by the CCB Coding Standards Philips Healthcare

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Change History

Revision	Date	Description
7.45	2020-09-28 14:12:55	Bram Stappers (TIOBE): - Redmine 26637: Fixed encoding portability issues for rules ERR#014, OLC#021, OOP#013, ENV33-C, and MSC33-C.
7.44	2020-09-24 22:55:00	Paul Jansen (TIOBE): - RTC 216366: Relaxed rule PCA#010.
7.43	2020-09-13 22:34:24	Paul Jansen (TIOBE): - RTC 213572: Removed rule INT#026.
7.42	2020-09-08 01:10:01	Paul Jansen (TIOBE): - RTC 213576: Removed rule CFL#020.
7.41	2020-09-04 21:35:21	Paul Jansen (TIOBE): - RTC 194061: Adjusted severity level of rule COM#002.
7.40	2020-09-01 23:53:52	Paul Jansen (TIOBE): - RTC 216368: Extended description of rule PCA#016.
7.39	2020-08-28 11:12:59	Paul Jansen (TIOBE): - RTC 50967: Adjusted severity level of rule PCA#011.
7.38	2020-08-16 23:21:01	Paul Jansen (TIOBE): - RTC 213575: Improved description of rule CFL#018.
7.37	2020-08-12 16:39:41	Paul Jansen (TIOBE): - RTC 210048: Removed rule PCA#007.
7.36	2020-08-12 16:33:18	Paul Jansen (TIOBE): - RTC 216364: Removed rule OOP#005.
7.35	2020-08-03 16:19:11	Paul Jansen (TIOBE): - RTC 212951: Improved rule OOP#013.
7.34	2020-08-03 15:57:31	Paul Jansen (TIOBE): - RTC 207470: Improved synopsis of rule OLC#018.
7.33	2020-07-10 23:33:55	Paul Jansen (TIOBE): - RTC 213577: Improved synopsis and description of rule OAL#018.
7.32	2020-07-09 20:42:47	Paul Jansen (TIOBE): - RTC 213578: Improved synopsis of rule OLC#005.
7.31	2020-07-08 00:06:54	Paul Jansen (TIOBE): - RTC 213578: Improved rule OLC#005.
7.30	2020-06-25 20:30:22	Paul Jansen (TIOBE): - RTC 204299: Introduced new rule CFL#026.
7.29	2020-06-21 00:22:04	Paul Jansen (TIOBE): - RTC 205613: Removed rule STY#004.
7.28	2020-06-19 00:08:44	Paul Jansen (TIOBE): - RTC 211070: Added descriptions to rules ERR#014 and ERR#016.
7.27	2020-06-07 21:54:55	Paul Jansen (TIOBE): - RTC 163635: Improved example of rule ERR#012.
7.26	2020-05-25 23:36:38	Paul Jansen (TIOBE): - RTC 66575: Improved rule INT#021.
7.25	2020-05-25 12:39:42	Paul Jansen (TIOBE): - RTC 161598: Introduced new rule CON#009.
7.24	2020-05-25 00:44:12	Paul Jansen (TIOBE): - RTC 197688: Extended rule STY#029.
7.23	2020-05-20 23:48:23	

		Paul Jansen (TIOBE): - RTC 201352: Removed rule CFL#022.
7.22	2020-04-14 10:42:55	Paul Jansen (TIOBE): - RTC 205392: Improved rule PCA#018 and removed rule OAL#017.
7.21	2020-03-21 00:09:50	Paul Jansen (TIOBE): - RTC 205339: Allowed constants for rule ORG#013.
7.20	2020-03-16 21:47:44	Paul Jansen (TIOBE): - RTC 196169: Introduced new rule CFL#025.
7.19	2020-02-27 18:30:11	Paul Jansen (TIOBE): - RTC 201898: Removed rule PCA#013.
7.18	2020-01-11 23:16:43	Paul Jansen (TIOBE): - RTC 167009: Removed rule OLC#008.
7.17	2020-01-04 20:50:54	Paul Jansen (TIOBE): - RTC 66583: Introduced new rule PCA#018.
7.16	2019-09-15 19:20:31	Paul Jansen (TIOBE): - RTC 97108: Removed rule OAL#005.
7.15	2019-08-10 15:58:57	Paul Jansen (TIOBE): - RTC 117848: Removed rule OLC#002.
7.14	2019-08-06 22:57:54	Paul Jansen (TIOBE): - RTC 185511: Improved rule PCA#007.
7.13	2019-07-20 22:57:25	Paul Jansen (TIOBE): - RTC 117847: Removed rule OAL#015.
7.12	2019-07-13 20:45:58	Paul Jansen (TIOBE): - RTC 86087: Improved description of rule OOP#013.
7.11	2019-05-25 22:01:51	Paul Jansen (TIOBE): - RTC 117850: Improved rule OLC#009.
7.10	2019-03-04 00:42:49	Paul Jansen (TIOBE): - RTC 66581: Improved description of rule PCA#001.
7.9	2019-03-03 10:55:12	Paul Jansen (TIOBE): - RTC 82230: Generalized rule GEN#002.
7.8	2019-02-24 23:25:43	Paul Jansen (TIOBE): - RTC 85004: Extended rule OLC#020 for base types.
7.7	2019-02-02 16:59:30	Paul Jansen (TIOBE): - RTC 124664: Improved description of rule ORG#011.
7.6	2019-01-27 14:27:28	Paul Jansen (TIOBE): - RTC 49970: Introduced new rule OAL#018.
7.5	2018-12-23 10:35:54	Paul Jansen (TIOBE): - RTC 117846: Removed rule CFL#013.
7.4	2018-12-14 23:32:31	Paul Jansen (TIOBE): - RTC 166687: Improved description of rule OLC#021.
7.3	2018-11-18 21:36:46	Paul Jansen (TIOBE): - RTC 103876: Introduced new rule OAL#017.
7.2	2018-10-27 22:22:08	Paul Jansen (TIOBE): - RTC 117844: Adjusted rule CFL#011.
7.1	2018-10-27 22:19:31	Paul Jansen (TIOBE): - RTC 84409: Removed rules OAL#007, OAL#008 and generalized rule OAL#011.
7.0	2018-10-27 22:04:42	Rob Douma (Philips)/Paul Jansen (TIOBE): - RTC 158629: Added security rules ARR38-C, DCL50-CPP,

		ENV33-C, ERR33-C, ERR54-CPP, EXP34-C, EXP53-CPP, EXT01-CPP, EXT02-CPP, EXT03-CPP, EXT04-CPP, EXT05-CPP, FIO30-C, FIO34-C, FIO37-C, FIO45-C, MEM50-CPP, MEM56-CPP, MSC30-C, MSC33-C, MSC51-CPP, STR31-C, STR32-C, STR38-C, STR50-CPP and STR51-CPP.
6.22	2018-09-03 00:59:29	Paul Jansen (TIOBE): - RTC 81705: Removed rule POR#036.
6.21	2018-08-12 23:32:24	Paul Jansen (TIOBE): - RTC 138794: Improved synopsis of rule OLC#021.
6.20	2018-08-12 23:18:33	Paul Jansen (TIOBE): - RTC 138794: Added new rule OLC#021.
6.19	2018-07-25 19:54:09	Paul Jansen (TIOBE): - RTC 154519: Adjusted severity level for rule ERR#012.
6.18	2018-07-25 17:18:56	Paul Jansen (TIOBE): - RTC 144003: Removed all references to the ISC++ and Ellemtel standards.
6.17	2018-07-24 19:08:47	Paul Jansen (TIOBE): - RTC 90693: Improved description of rule PCA#002.
6.16	2018-04-10 02:12:01	Paul Jansen (TIOBE): - RTC 76091: Introduced new rule PCA#017.
6.15	2018-03-03 15:38:46	Paul Jansen (TIOBE): - RTC 137163: Improved synopsis, description and definition of rule INT#027.
6.14	2018-02-15 14:26:35	Paul Jansen (TIOBE): - RTC 117838: Improved synopsis, description and definition of rule CFL#019.
6.13	2018-01-04 00:07:18	Paul Jansen (TIOBE): - RTC 127759: Improved the description of rule OLC#005.
6.12	2017-12-30 01:43:58	Paul Jansen (TIOBE): - RTC 117843: Improved synopsis, description and definition of rule CFL#009.
6.11	2017-06-24 18:54:13	Paul Jansen (TIOBE): - RTC 47077: Improved synopsis and description of rule CFL#014.
6.10	2017-06-24 18:29:57	Paul Jansen (TIOBE): - RTC 29011: Replaced the term "include file" by "header file".
6.9	2017-05-15 00:37:25	Paul Jansen (TIOBE): - RTC 81703: Added exception to rule NAM#008. - RTC 82228: Improved synopsis of rule CFL#020.
6.8	2017-05-13 18:18:52	Paul Jansen (TIOBE): - RTC 109536: Improved rule ERR#012.
6.7	2017-05-04 00:55:08	Paul Jansen (TIOBE): - RTC 66575: Improved rule INT#021.
6.6	2017-03-29 12:36:46	Paul Jansen (TIOBE): - RTC 47083: Removed rule ERR#002. - RTC 63275: Added exception to rule CFL#002. - RTC 66579: Improved description of rule OOP#001. - RTC 66582: Improved description of rule STY#020. - RTC 84425: Removed rule POR#035. - RTC 95342: Added rule OAL#015.
6.5	2017-03-21 01:53:25	Paul Jansen (TIOBE):

		- RTC 82229: Changed synopsis of rule CON#008.
6.4	2017-02-20 21:47:30	Paul Jansen (TIOBE): - RTC 88854: Extended rule INT#027.
6.3	2017-01-26 22:23:00	Paul Jansen (TIOBE): - RTC 91090: Removed rule STY#018.
6.2	2016-12-24 22:24:20	Paul Jansen (TIOBE): - RTC 94331: Changed all occurrences of NULL by nullptr.
6.1	2016-02-10 21:06:20	Paul Jansen (TIOBE) - RTC 72958: Resurrected rule CFL#024.
6.0	2015-08-27 16:57:03	Paul Jansen (TIOBE) - RTC 45178: Added rules OAL#011, OAL#012, OAL#013. - RTC 45178: Improved rule OAL#008 to conform to RAII. - RTC 47076: Extended rule INT#027 with keyword override. - RTC 47081: Added exception of smart pointers to CFL#011 and CON#007. - RTC 47085: Revised rule ERR#006 to conform to C++11. - RTC 47087: Removed rule ERR#010. - RTC 49185: Introduced the swap idiom for rule OLC#002. - RTC 50551: Improved rule CFL#018 to use the C++11 range-based for loop. - RTC 50555: Changed rule POR#032 to use the C++11 nullptr keyword. - RTC 50557: Improved rule OAL#004 to be compliant with C++11. - RTC 50558: Changed rule OLC#001 to conform to C++11. - RTC 50559: Removed rule OOP#006. - RTC 50560: Changed rule STY#020 to conform to C++11. - RTC 50562: Introduced rule PCA#016.
5.8	2015-03-22 23:15:51	Paul Jansen (TIOBE) - RTC-51109: improved description of rule OLC#004.
5.7	2015-01-05 00:05:51	Paul Jansen (TIOBE) - RTC-46173: removed rule OAL#006 and extended rule OAL#007.
5.6	2014-11-10 22:45:52	Paul Jansen (TIOBE) - Made rule INT#006 checkable.
5.5	2014-10-05 19:18:11	Paul Jansen (TIOBE) - Fixed typos in description of rule CON#004.
5.4	2014-09-24 20:35:25	Paul Jansen (TIOBE) - RTC-42457: added rule ORG#012.
5.3	2014-03-30 23:20:37	Paul Jansen (TIOBE) - RTC-30209: removed rule INT#020.
5.2	2013-11-17 15:58:31	Paul Jansen (TIOBE) - Made rule CFL#019 checkable.
5.1	2013-11-12 14:21:18	Paul Jansen (TIOBE) - RTC-11329: made rule CON#007 checkable. - RTC-19317: allowed UK English as well for rule COM#003.
5.0 (Authorized)	2013-01-22 14:14:41	Paul Jansen (TIOBE) - Processed review comments. - RTC-2529: added rule CON#008. - RTC-11836: removed functions from rule PRE#001 and changed severity level.
4.1	2012-11-01 10:31:50	Paul Jansen (TIOBE) - RTC-1424: changed rule CFL#016 to use cyclomatic complexity.

		- RTC-6472: added a rationale to rule POR#025. - RTC-6982: added an exception for "stdafx.h" to rule ORG#005. - RTC-6983: removed rule PCA#012. - RTC-6985: added a rationale to rule CON#006. - RTC-8478: introduced new rule POR#038. - RTC-9098: added built-in type conversions to rule CON#004 and improved its description. - RTC-10095: removed the following rules as discussed by the C++ coding standards refresh team: CFL#003, CFL#008, CFL#010, CFL#012, CFL#015, CFL#021, CFL#023, CFL#024, COM#001, COM#004, COM#007, CON#003, CON#005, ERR#004, ERR#007, ERR#008, ERR#009, ERR#013, GEN#001, GEN#003, INT#003, INT#004, INT#005, INT#007, INT#009, INT#012, INT#013, INT#016, INT#017, INT#018, INT#019, INT#024, INT#025, NAM#001, NAM#004, NAM#006, NAM#007, NAM#009, NAM#010, NAM#011, NAM#012, NAM#015, OAL#001, OAL#010, OLC#007, OLC#011, OLC#013, OLC#014, OOP#010, OOP#012, OOP#014, OOP#015, OOP#016, OPT#003, ORG#008, PCA#004, PCA#014, PCA#015, POR#023, POR#024, POR#026, POR#027, POR#034, STA#003, STY#001, STY#003, STY#005, STY#007, STY#008, STY#009, STY#010, STY#011, STY#012, STY#013, STY#014, STY#015, STY#016, STY#019, STY#021, STY#023, STY#026, STY#027, STY#030, STY#031. - RTC-10095: Adapted the following rules as discussed by the C++ coding standards refresh team: CFL#014, CFL#016, CFL#018, COM#002, CON#006, INT#006, INT#008, INT#014, OLC#009, OLC#010, OLC#012, OLC#019, OOP#002, OOP#005, OOP#006, OOP#007, ORG#007, PCA#009, POR#035.
4.0 (Authorized)	2012-04-25 18:57:52	Issam Tlili Reworked version after review of version 3.1
3.1	2012-02-14 10:38:52	Issam Tlili WI-952: adapted description of OLC#012. WI-953: in OLC#016 aligned the C++ implementation with the C# implementation. WI-970: new C++ rule STY#031. WI-984: exclude namespaces from rule STY#009. WI-988: relax the allowed extension for include files. WI-992: in POR#003 changed severity level to level 4. WI-993: adapted the description in CON#001, in CON#003 removed the second part of the description, it is already mentioned in CON#001. WI-1002: added exception for const array definition in OLC#009. WI-1040: adapted ERR#005 and CON#004. WI-1057: adapted STY#012 and STY#013. WI-1058: adapted rule for ternary operators. WI-1003: added description for OLC#018. WI-954: new rule INT#030. WI-1079: adapted description of OAL#005, and removed reference to OLC#013. WI-1086: adapted INT#013. WI-1415: adapted OLC#019. WI-1091: removed OLC#015, in CFL#023 removed reference to OLC#015. WI-1239: new rule NAM#015, removed NAM#003, NAM#005, NAM#014 and STY#022.

3.0.1	2011-11-22 12:27:40	Issam Tlili Draft, preparations for 3.1
3.0 (Authorized)	2010-08-27 11:55:56	Issam Tlili Reworked version after review of version 2.4
2.4	2010-07-08 11:28:10	Issam Tlili Reworked version after review of version 2.2 Set rule CFL#024 to the right Category (CFL).
2.3	2010-07-08 11:08:33	Issam Tlili Reworked version after review of version 2.2 Rule STY#031 moved to CFL#24
2.2	2010-05-17 16:15:33	Issam Tlili IM-TA00006832: Adapt STY#024 for include naming. IM-TA00006738: Clarified description in POR#003. IM-TA00006714: New rule STY#030 for useless statements. IM-TA00006103: New rule OLC#020; don't pass member variables to the base class in the constructor. IM-TA00005056: Rule moved from POR#013 to GEN#003, and adapted synopsis. IM-TA00004284: POR#018; severity changed to level 1. IM-TA00002758: New rule OOP#018 for overriding const member function. IM-TA00002632: New rule INT#029; Use built-in boolean type where possible. IM-TA00002631 and IM-TA00002630: New rule CON#007; Do not convert implicitly from a boolean type to a non-boolean type.
2.1	2010-05-17 15:38:58	Issam Tlili IM-TA00006832: Adapt STY#024 for include naming IM-TA00006738: Clarified description in POR#003 IM-TA00006714: Add rule for useless statements
2.0 (Authorized)	2009-04-15 16:12:53	Vic Henderikx Reworked version after review of version 1.7
1.7	2009-03-18 14:06:29	Vic Henderikx IM-TA00004418: PCA#001: also forbid functions using old C-style allocation IM-TA00004585: NAM#005: allow m_PascalCase for member variables IM-TA00004654: removed NAM#013; added ORG#011: always namespaces IM-TA00005570: Added OAL#010: don't overload new or delete IM-TA00005054: CON#004 now level 1; added CON#006 reinterpret_cast; updated CON#001 (removed reinterpret_cast) IM_TA00004313: OLC#017 also for pointers to all types IM_TA00004589: ERR#017 added: catch_all must rethrow IM_TA00004789: ORG#001: also allow #pragma once IM_TA00004884: STY#010: remove part about repeating "virtual" or "static" in comment IM_TA00004891: CFL#021: non-simple types iso user defined types IM-TA00005040: OOP#008: "in a class" --> "in a class or in a namespace" IM-TA00004398: INT#020: use * or & for class-typed arguments; now also for structs IM-TA00005407: new rule POR#036: don't use parenthesis when declaring a class variable using default constructor

		IM-TA00004890: removed OLC#013 about not giving values to simple types in destructors
1.6.1	2008-11-26 19:48:39	Vic Henderikx PCA#001: also avoid functions using C-style allocators
1.6	2008-05-22 12:15:47	Vic Henderikx Added PCA#013 (avoid trigraphs) POR#010 is not checked by TICS (decision CCB)
1.5	2008-05-21 13:09:26	Changed PMS to Philips Healthcare for rule GEN#002
1.4	2008-04-25 14:17:26	Daan van der Munnik changed PMS to Philips Healthcare
1.3	2008-04-25 12:57:12	Vic Henderikx IM-TA00003860 (CON#002 set from level 1 to level 2) IM-TA00004003 (Removed STY#028) IM-TA00003640 (Rephrased OLC#015) IM-TA00004004 (INT#002: Removed paragraph about naming of members) IM-TA00004005 (Removed INT#016)
1.2	2008-04-25 12:38:44	Update Philips Logo
1.1	2008-04-25 11:19:48	Initial Import from firebird to oracle

Introduction

Purpose

This document defines the Philips Healthcare C++ Coding Standard. The coding standard consists of rules and recommendations for C++ code written and/or maintained by the Philips Healthcare SW departments. It replaces department specific C++ coding standards [Schaick] and [Tongeren].

Scope

The coding standard is based on the rules and recommendations of several sources. These include "Industrial Strength C++" [ISC++], "Philips Medical Systems C Coding Standard" [Hatton], "PMS-MR C++ Coding Standard" [Tongeren], and "CIS C++ Development Guidelines" [Schaick].

The procedure to be followed when a rule must be broken is outside the scope of this document. This procedure should be defined at project or department level.

This coding standard specifies rules

- to prevent coding errors
- to prevent coding using obsolete or deprecated language features
- to prevent coding using undefined or implementation defined language features
- to achieve a certain consistency in coding style

The Philips Healthcare C++ coding standard does not attempt to teach how to design effective C++ code. It also does not categorically rule out any programming idioms that C++ is designed to support. Background information on the rationale for C++ language design decisions are documented in [StroustrupDE]. Applying C++ effectively is the subject of e.g. [Stroustrup], which also lists many useful advises at the end of each chapter.

Document Layout

The coding standard document consists of a set of rules. Rules are grouped together logically into so called categories. Each chapter deals with one category. A rule description contains the following items:

- **Synopsis**. This is a brief description of the rule.
- **Language**. The language item indicates for what language(s) the rule is applicable.
- **Severity Level**. All rules in this coding standard have a level assigned to it, ranging from 1 to 10. Level 1 concerns the most severe issues (program errors), whereas level 10 consists of the least important rules (style issues). The rules have been distributed evenly over the available levels to be able to focus on a certain limited set of violations.
- **Category**. This indicates to what category a rule belongs.
- **Description**. The description explains the rule in more detail. Sometimes it also contains a justification of the rule, possible exceptions and code examples.
- **Literature References**. This section contains references to the origin of the rule. It can also be used for further reading.

Class Interface

This chapter concerns the public interface of classes. The class interface is the most important part of the class. It is the contract to be used by the clients of the class.

Rules

INT#001	Non-copy-constructors that can be called with one argument shall be declared as explicit
INT#002	Declare non-constant data members private
INT#006	A member function that should not change the state of the object shall be declared const
INT#008	Use constant references (const &) instead of call-by-value, unless using a basic data type, a simple object or a pointer
INT#010	Use operator overloading sparingly and in a uniform manner
INT#011	If you overload one of a closely related set of operators, then you should overload the whole set and preserve the same invariants that exist for built-in types
INT#014	Use a parameter of pointer type if the function stores the address or passes it to a function that does
INT#015	All variants of an overloaded member function shall be used for the same purpose and have similar behavior
INT#021	Pass arguments of class types by reference or pointer if the class is meant as a public base class
INT#022	A pointer or reference parameter should be declared const if the function does not change the object bound to it
INT#023	The copy constructor and the copy assignment operator shall always have a const reference as a parameter
INT#027	If you override one of the base class's virtual functions, then you shall use the "override" keyword
INT#028	Supply default arguments with the function's declaration, not with the function's definition
INT#029	Use built-in boolean type where possible
INT#030	Do not misuse a pointer when an array is requested

Rule INT#001

Synopsis: Non-copy-constructors that can be called with one argument shall be declared as explicit
Language: C++
Level: 2
Category: Class Interface

Description

When using constructors with single arguments, these constructors need to be declared with the keyword *explicit*. This keyword prevents implicit type conversions for the arguments to the constructor.

```
explicit CTestClass(SHORT nValue);
```

This syntax prevents an implicit conversion from the *float* argument in the next example, which can cause undesirable side effects.

```
CTestClass TestClass = 3.2; // Does not work
```

This rule does not apply to copy-constructors, because they do not define type conversions. This rule does apply to constructors with N arguments, with N or N-1 default values. See also [OLC#007] and [CON#001].

Rule INT#002

Synopsis: Declare non-constant data members private
Language: C++
Level: 2
Category: Class Interface

Description

As a consequence when data needs to be accessed, be it by a derived class or an external class, protected or public Set and Get methods (Accessors) need to be implemented for these private data members. This way, the class implementation is separated from the class interface.

Note that declaring data members protected is usually a design error, and easily becomes a maintenance problem. See the convincing justification in section 15.3.1.1 of [Stroustrup]. If you want to model a data structure, with public access to its data members, then use a struct type.

Exception: *constant* data members are often used as constant values with no need of information hiding. In this case it's allowed to declare these members public or protected.

Rule INT#006

Synopsis: A member function that should not change the state of the object shall be declared const
Language: C++
Level: 2
Category: Class Interface

Description

It is possible that a const method requires non-const access to some of the data members, without changing the objects state. For example, when a lock variable (i.e. critical section) is used to synchronize access to other data members. In this case, the lock variable should be made *mutable*. See also [CON#005].

Rule INT#008

Synopsis: Use constant references (const &) instead of call-by-value, unless using a basic data type, a simple object or a pointer
Language: C++
Level: 7
Category: Class Interface

Description

Method/function arguments are invoked according to call-by-value. This means that values are copied with invocations to constructors and destructors as a result. This does no harm for basic data types, simple objects and pointers, but reduces performance for complex objects. When arguments are passed using references (call-by-reference), only the reference itself is copied.

```
void SetComplexObject(const ComplexObject sName);    // Less efficient.
void SetComplexObject(const ComplexObject& rsName);  // More efficient.
```

A simple object is defined as an object that has at most 5 leave basic data types. E.g. both

```
struct Coordinate {
  int x;
  int y;
  int z;
};
```

and

```
class Product {
  ...
private:
  Coordinate loc;
  float price;
};
```

are considered simple objects in this context.

Rule INT#010

Synopsis: Use operator overloading sparingly and in a uniform manner
Language: C++
Level: 9
Category: Class Interface

Description

A disadvantage in overloading operators is that it is easy to misunderstand the meaning of an overloaded operator (if natural semantics are not used). Do not use it if it can easily give rise to misunderstanding. Keep to the old advice: "do as the int's do".

Rule INT#011

Synopsis: If you overload one of a closely related set of operators, then you should overload the whole set and preserve the same invariants that exist for built-in types
Language: C++
Level: 9
Category: Class Interface

Description

When an expression is using one of the operators, it is expected to work with the opposite operator as well.

Rule INT#014

Synopsis: Use a parameter of pointer type if the function stores the address or passes it to a function that does
Language: C++
Level: 9
Category: Class Interface

Description

A pointer is also required if the parameter can refer to no object, i.e. have the value nullptr. A reference cannot model this, because it is always linked to the same object with which it was initialized.

Rule INT#015

Synopsis: All variants of an overloaded member function shall be used for the same purpose and have similar behavior
Language: C++
Level: 2
Category: Class Interface

Description

Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments. If not used properly (such as using functions with the same name for different purposes), they can, however, cause considerable confusion.

Rule INT#021

Synopsis: Pass arguments of class types by reference or pointer if the class is meant as a public base class
Language: C++
Level: 9
Category: Class Interface

Description

What is meant is that the function should be able to use the class interface of the passed object, which may actually be of a derived type, in a polymorphic way. This is not possible if the object is passed by value.

Note that such passing by value is not even possible if the argument refers to an abstract class.

Rule INT#022

Synopsis: A pointer or reference parameter should be declared const if the function does not change the object bound to it
Language: C++
Level: 4
Category: Class Interface

Description

Note that what is meant is: the pointee, instead of the pointer, should be declared const if appropriate. See also [INT#020].

```
void f(const int* ptr);  // okay: pass ptr to a const int
void f(const int& ref);  // okay: pass ref to a const int
```

Rule INT#023

Synopsis: The copy constructor and the copy assignment operator shall always have a const reference as a parameter
Language: C++
Level: 5
Category: Class Interface

Rule INT#027

Synopsis: If you override one of the base class's virtual functions, then you shall use the "override" keyword
Language: C++
Level: 2
Category: Class Interface

Description

It is not possible to remove the virtual-ness of an inherited virtual function: any virtual function overrides itself. This rule makes explicit that the function that overrides a virtual function is virtual too, which it would also have been if the overriding function were not declared as such.

By adding the "override" keyword you express your intention that this function is an override of a function in the parent class. This is also checked by the compiler. If you make a mistake (see example below) the compiler will issue an error to indicate that the function doesn't override another function:

```
class base
{
  public:
    virtual int foo(float x) = 0;
};


class derived: public base
{
   public:
     virtual int foo(float x) { ... } // before C++11, not accepted any more
     int foo(float x) override { ... } // C++11
```

```
}

class derived2: public base
{
   public:
     int foo(int x) override { ... }  // won't compile!
};
```

Rule INT#028

Synopsis: Supply default arguments with the function's declaration, not with the function's definition
Language: C++
Level: 2
Category: Class Interface

Description

Note that the use default arguments is generally not recommended: see [PCA#007].

Rule INT#029

Synopsis: Use built-in boolean type where possible
Language: C++
Level: 6
Category: Class Interface

Description

For portability and readability reasons, use built-in boolean type where possible. All C++ standard conforming compilers support bool type.

Rule INT#030

Synopsis: Do not misuse a pointer when an array is requested
Language: C++
Level: 2
Category: Class Interface

Description

When a pointer is used as reference to an array, it shall refer to an array of the correct type.

Example:

```
void Foo( int a[], int size )
{
    // do something with a;
    for (int i = 0; i < size; i++)
    {
```

```
        a[i] = i;
    }
}

int ar[] = {0,0,0};
Foo( ar, sizeof(ar)/sizeof(ar[0]) ); // Ok
...
int* ptr = &ar[1];
Foo( ptr, sizeof(ar)/sizeof(ar[0]) ); //Not allowed: passing pointer ptr in Foo causes out
```

Comments

This chapter concerns comments in the code. Comments highly increase the readability and maintainability of the code.

Rules

COM#002	All files must include copyright information
COM#003	All comments are to be written in English
COM#005	Do not leave commented-out code in the source file

Rule COM#002

Synopsis: All files must include copyright information
Language: C++
Level: 2
Category: Comments

Description

Each file should at least contain the word "Copyright" or "(c)", a reference to the name of the company and a year.

Rule COM#003

Synopsis: All comments are to be written in English
Language: C++
Level: 10
Category: Comments

Description

Fewer programmers may understand comments written in your native language (unless your native language is English). Note that comments shall especially not contain any Greek or other symbolic non-ASCII tokens, because they are not portable and cannot be handled properly by the lexical analyzers of certain tools.

Rule COM#005

Synopsis: Do not leave commented-out code in the source file
Language: C++
Level: 9
Category: Comments

Description

When commented-out code is left in the source file it may later be unclear if the code is still up-to-date. It also may result in extra work when during maintenance the commented-out code is kept up-to-date, even when the code may never be used again.

Control Flow

This chapter concerns control flow through control statements, such as if, for, while, do, switch and goto.

Rules

CFL#001	Statements following a case label shall be terminated by a statement that exits the switch statement
CFL#002	All switch statements shall have a default label as the last case label
CFL#004	Do not use goto
CFL#005	Do not access a modified object more than once in an expression
CFL#006	Do not apply sizeof to an expression with side-effects
CFL#007	Do not change a loop variable inside a for loop block
CFL#009	Never use continue in a nested loop
CFL#011	The test condition in control statements shall be a non-assignment expression
CFL#014	Do not return from unexpected locations
CFL#016	Do not have overly complex functions
CFL#017	Do not make explicit comparisons to true or false
CFL#018	Use range-based for loops if possible
CFL#019	Use explicit parentheses when using multiple different operators in an expression
CFL#024	A statement must have a side-effect, i.e., it must do something.
CFL#025	Use a reference to a range based loop "auto" variable if it is modified in its body
CFL#026	Use conditions to avoid spurious wakes

Rule CFL#001

Synopsis: Statements following a case label shall be terminated by a statement that exits the switch statement
Language: C++
Level: 2
Category: Control Flow

Description

If the code which follows a case label is not terminated by break, the execution continues after the next case label. A missing break statement can easily be overlooked resulting in erroneous code. Usually a break will be used to exit the switch statement. Be careful when using a return to exit the switch statement as in the example of ISC++: see [CFL#020].

Note that it is allowed to 'fall through' case labels: see [CFL#002].

Rule CFL#002

Synopsis: All switch statements shall have a default label as the last case label
Language: C++
Level: 2

Category: Control Flow

Description

Example:

```
switch (c)
{
case c0:        // fall-through
case c1:
    {
        x;
        break;
    }
case c2:
    {
        y;
        break;
    }
default:        // always 'default' at end, even if no statement follows
    {
        z;      // possibly an assertion that this should not be reached
        break;  // leave break: in case this becomes a non-default case
    }
}
```

This rule doesn't hold for enumerations if all enum values are explicitly used as case labels in the switch statement. If one of the enum values of the enumeration is missing in the switch body this rule will trigger as well. This is to make sure one doesn't forget the add new enum values to the switch statement in case the enumeration is extended.

```
enum Color { red, green, blue };
Color r = red;
switch (r)
{
    case red  : std::cout
```

Rule CFL#004

Synopsis: Do not use goto
Language: C++
Level: 7
Category: Control Flow

Description

The ISC++ rule is adapted to clarify that the goto statement is never allowed, as is explained in ISC++. This implies that labels are only allowed in switch statements.

Rule CFL#005

Synopsis: Do not access a modified object more than once in an expression
Language: C++
Level: 1

Category: Control Flow

Description

The evaluation order of sub-expressions within an expression is undefined. See also [POR#029].

Example:

```
v[i] = ++c;     // right
v[i] = ++i;     // wrong: is v[i] or v[++i] being assigned to?
i = i + 1;      // right
i = ++i + 1;    // wrong: i is modified twice in an expression
```

Rule CFL#006

Synopsis: Do not apply sizeof to an expression with side-effects
Language: C++
Level: 1
Category: Control Flow

Description

The operand of sizeof is not evaluated, so any intended side-effects will not occur.

Example:

```
size_t Size = sizeof(nCount++); // Wrong: nCount not changed!
```

Rule CFL#007

Synopsis: Do not change a loop variable inside a for loop block
Language: C++
Level: 2
Category: Control Flow

Rule CFL#009

Synopsis: Never use continue in a nested loop
Language: C++
Level: 9
Category: Control Flow

Description

The "continue" statement is used to force the next iteration of a loop to take place. This can help to make loops in code more readable. Compare for instance the following code containing "continue" statements:

```
for(object_id_t id : proc_list)
{
```

```
  object* obj = find_object(id);
  if(!obj)
    continue;

  if(obj->is_active())
    continue;

  time_t elapsed = now() - obj->begin;
  if(elapsed < timeout)
    continue;

  ...
}
```

with the semantically equivalent code without "continue" statements:

```
for(object_id_t id : proc_list)
{
  object * obj = find_object(id);
  if(obj)
  {
    if(!obj->is_active())
    {
      time_t elapsed = now() - obj->begin;
      if(elapsed >= timeout)
      {
        ...
      }
    }
  }
}
```

The second example is less readable because nested "if" statements are needed.

But the "continue" statement also has a downside. Since "continue" is tied to its directly enclosing loop, its meaning can change unexpectedly if the "continue" statement is moved. The compiler will not warn you about this. Consider for example the following code:

```
for (int i = 0; i < max; i++) {
  ...
  for (int j = 0; j < max; j++) {
    ...
    some code
      continue; // somewhere a little bit hidden
    some other code
    ...
  }
}
```

and suppose some lines of code are moved to the outer loop:

```
for (int i = 0; i < max; i++) {
  ...
  some code
    continue; // somewhere a little bit hidden
  some other code
  ...
  for (int j = 0; j < max; j++) {
    ...
  }
}
```

Now all of a sudden the inner loop is not executed any more if the "continue" statement is reached. If this is not intended it might take a lot of debugging effort to understand why this happened. So it is better to avoid "continue" statements in nested loops.

Rule CFL#011

Synopsis: The test condition in control statements shall be a non-assignment expression
Language: C++
Level: 9
Category: Control Flow

Description

The test condition shall not be an assignment expression. This avoids mistaking assignments for comparisons.

Example:

```
bool b1 = true;
bool b2 = true;
int i = 123;

if (i != 0) // right: expression is a non-assignment
if (i = 123) // wrong: probably, (i == 123) was intended
if (b1 = (b1 && b2)) // wrong: even if this might seem convenient in some cases
```

Rule CFL#014

Synopsis: Do not return from unexpected locations
Language: C++
Level: 9
Category: Control Flow

Description

It is harder to understand a function if, reading it at the bottom, you are unaware of the possibility that it returned somewhere above. So multiple returns in a function should be avoided, if possible. However, in some situations it is perfectly legal to have extra returns. Examples are:

- precondition checks that fail at the beginning of a function
- loops that should terminate as soon as some condition is met (e.g. return the first element larger than 10 in a list)

Rule CFL#016

Synopsis: Do not have overly complex functions
Language: C++
Level: 4
Category: Control Flow

Description

If a function is too complex, it can be difficult to comprehend. The cyclomatic complexity of a function, i.e. the number of linearly independent paths through a function, should not exceed the agreed maximum. A function with a cyclomatic complexity less than 5 is considered to be ok. However, if its cyclomatic complexity is higher than 10, refactoring is recommended.

Rule CFL#017

Synopsis: Do not make explicit comparisons to true or false
Language: C++
Level: 9
Category: Control Flow

Description

It is usually bad style to compare a bool-type expression to true or false.

It is even dangerous to compare a non-bool-type expression to true: that might unexpectedly evaluate to false.

Example:

```
while (condition == false)      // wrong; bad style
while (condition == true)       // wrong; possibly dangerous
while (boolean_condition)       // okay if not an assignment, see CFL#011.
```

Rule CFL#018

Synopsis: Use range-based for loops if possible
Language: C++
Level: 4
Category: Control Flow

Description

Most iterators and C-style index based for loops can be rewritten as range-based loops. Range-based for loops are less error prone, so if possible, use them.

Wrong Example:

```
std::vector<char> myString;
for (auto iter = myString.begin(); iter != myString.end(); iter++)
```

Correct Example:

```
std::vector<char> myString;
for (auto & value : myString)
```

Rule CFL#019

Synopsis: Use explicit parentheses when using multiple different operators in an expression
Language: C++
Level: 9
Category: Control Flow

Description

Without parentheses, it is easy to make mistakes when relying on precedence rules. Comparison and unary operators are excluded from this rule.

Example:

```
if ( a || b && c || d )   // wrong: incorrect reliance on precedence
if ((a || (b && c) || d) // right: parentheses for correct evaluation
```

Rule CFL#024

Synopsis: A statement must have a side-effect, i.e., it must do something.
Language: C++
Level: 1
Category: Control Flow

Description

Example:

```
i == 1; // WRONG - operator == has no effect
i;      // WRONG - but legal!
i++;    // RIGHT - does something
```

Rule CFL#025

Synopsis: Use a reference to a range based loop "auto" variable if it is modified in its body
Language: C++
Level: 2
Category: Control Flow

Description

Consider the following code

```
std::vector<int> ints{ 10, 20, 30 };
for(auto i : ints) {
  if (i == 30) {
    i = 40;
  }
}
do something with "ints"
```

Then after the for loop the "ints" variable will still be unchanged! Instead a reference to the "auto" variable should be used:

```
std::vector<int> ints{ 10, 20, 30 };
for(auto& i : ints) {
  if (i == 30) {
    i = 40;
  }
}
do something with "ints"
```

Rule CFL#026

Synopsis: Use conditions to avoid spurious wakes
Language: C++
Level: 3
Category: Control Flow

Description

A spurious wakeup happens when a thread wakes up from waiting on a condition variable that is been signaled, only to discover that the condition it was waiting for isn't satisfied. It is called spurious because the thread has seemingly been awakened for no reason. But spurious wakeups don't happen for no reason, they usually happen because in between the time when the condition variable was signaled and when the waiting thread finally ran, another thread ran and changed the condition. There was a race condition between the threads, with the typical result that sometimes, the thread waking up on the condition variable runs first, winning the race, and sometimes it runs second, losing the race.

Spurious wakes can cause some unfortunate bugs, which are hard to track down due to the unpredictability of spurious wakes. These problems can be avoided by using a predicate that is packaged as a function or function object, using the predicated overloads of wait(), wait_for() and wait_until().

Wrong example:

```
std::condition_variable cv;
std::mutex cv_m;
int i = 0;

void waits()
{
    std::unique_lock lk(cv_m);
    std::cerr  lk(cv_m);
        std::cerr  lk(cv_m);
        i = 1;
        std::cerr
```

The wait call should have one extra predicate argument:

```
std::condition_variable cv;
std::mutex cv_m;
int i = 0;

void waits()
{
    std::unique_lock lk(cv_m);
    std::cerr  lk(cv_m);
        std::cerr  lk(cv_m);
```

```
i = 1;
std::cerr
```

Conversions

This chapter concerns the conversion of objects from one type to another. Wrong usage of conversions may result in less robust code.

Rules

CON#001	Make unsafe type conversions explicit rather than implicit
CON#002	Do not cast away const
CON#004	Use the new cast operators (static_cast, const_cast, dynamic_cast, and reinterpret_cast) instead of the C-style casts
CON#006	Prefer static_cast over reinterpret_cast if possible
CON#007	Do not convert implicitly from a boolean type to a non-boolean type, and vice versa.
CON#008	Don't take the address of an array
CON#009	Don't compare an address to null

Rule CON#001

Synopsis: Make unsafe type conversions explicit rather than implicit
Language: C++
Level: 2
Category: Conversions

Description

If casting is required, make it explicit and use the following C++ cast operators:

- const_cast: to remove the const, volatile, and __unaligned attributes.
- dynamic_cast: for conversion of polymorphic types, where class hierachy navigation is unavoidable.
- static_cast: for conversion of nonpolymorphic types.

This rule does not only apply to casting: it also applies to unsafe conversions by means of constructors, conversion operators, and conversion functions. See further [INT#001], [INT#017], [POR#022].

Rule CON#002

Synopsis: Do not cast away const
Language: C++
Level: 2
Category: Conversions

Description

A const variable is not meant to be changed. Note that it should be possible for a compiler to allocate constants in read-only memory.

Rule CON#004

Synopsis: Use the new cast operators (static_cast, const_cast, dynamic_cast, and reinterpret_cast) instead of the C-style casts
Language: C++
Level: 1
Category: Conversions

Description

Let's assume these:

```
class CMyClass : public CMyBase {...};
class CMyOtherStuff {...} ;

CMyBase  *pSomething; // Filled somewhere
```

Now, these two are compiled the same way:

```
CMyClass *pMyObject;
pMyObject = static_cast<CMyClass*>(pSomething); // Safe; as long as we checked

pMyObject = (CMyClass*)(pSomething); // Same as static_cast. Safe; as long as we checked
```

However, let's see this almost identical code:

```
CMyOtherStuff *pOther;
pOther = static_cast<CMyOtherStuff*>(pSomething); // Compiler error: Can't convert

pOther = (CMyOtherStuff*)(pSomething); // No compiler error. Same as reinterpret_cast ar
```

As you can see, there is no easy way to distinguish between the two situations without knowing a lot about all the classes involved.

The second problem is that the C-style casts are too hard to locate. In complex expressions it can be very hard to see a C-style cast. It is virtually impossible to write an automated tool that needs to locate C-style casts (for example a search tool) without a full blown C++ compiler front-end. On the other hand, it's easy to search for "static_cast

```
pOther = reinterpret_cast<CMyOtherStuff*>(pSomething);
      // No compiler error.
      // but the presence of a reinterpret_cast is
      // like a Siren with Red Flashing Lights in your code.
      // The mere typing of it should cause you to feel VERY uncomfortable.
```

That means that, not only are C-style casts more dangerous, but it's a lot harder to find them all to make sure that they are correct.

See also [CON#006] for a discussion about the new cast operators.

Since conversion functions for built-in types such as behave exactly the same as C style casts, these aren't allowed either:

```
a = int(b); // Wrong, is the same as a = (int) b;
```

Exception to this rule: Using (void) to cast-away the return value is allowed.

Rule CON#006

Synopsis: Prefer static_cast over reinterpret_cast if possible
Language: C++
Level: 4
Category: Conversions

Description

A static_cast is a cast from one type to another for which a known method for conversion is available. For example, you can static_cast an int to a char because such a conversion is meaningful. However, you cannot static_cast an int* to a double*, since this conversion only makes sense if the int* has somehow been mangled to point at a double*.

A reinterpret_cast is a cast that represents an unsafe conversion that might reinterpret the bits of one value as the bits of another value. For example, casting an int* to a double* is legal with a reinterpret_cast, though the result is unspecified. Similarly, casting an int to a void* is perfectly legal with reinterpret_cast, though it's unsafe.

Neither static_cast nor reinterpret_cast can remove const from something. You cannot cast a const int* to an int* using either of these casts. For this, you would use a const_cast.

In general, you should always prefer static_cast for casts that should be safe. If you accidentally try doing a cast that isn't well-defined, then the compiler will report an error. Only use reinterpret_cast if what you're doing really is changing the interpretation of some bits in the machine.

Rule CON#007

Synopsis: Do not convert implicitly from a boolean type to a non-boolean type, and vice versa.
Language: C++
Level: 3
Category: Conversions

Description

Example:

```
int i = 5, j = 4, k = 3, l = 2;

bool b = (i > j);              // right
int a = (i > j);                 // wrong: implicit conversion from bool to int
int a = ((i > j) + (k > l)) // wrong: implicit conversion from bool to int
```

Exception:

Since smart pointers are designed according to the "safe bool idiom" (they behave correctly in boolean context), it is also allowed to use a smart pointer in a boolean context, e.g.

```
shared_ptr<A> a_sp = someFunctionReturningSharedPtr();
bool my_bool = a_sp;   // right: smart pointers are implicitly of boolean type
```

Rule CON#008

Synopsis: Don't take the address of an array
Language: C++
Level: 1
Category: Conversions

Description

One should take care that an array variable is a pointer to the first element of the array. This can lead to subtle unintended incorrect usage and crashes as shown in the example below:

```
const DWORD mv_dwReadEventLogBufferSize = 64 * 1024;

// All records from the event log can be read now.
BYTE abyBuffer[mv_dwReadEventLogBufferSize];

// EVENTLOGRECORD defined in winnt.h
// Compiles fine but is wrong. Variable 'pEventLogRecord' contains one indirection too
EVENTLOGRECORD* pEventLogRecord = (EVENTLOGRECORD*) &abyBuffer;
```

It should have been as follows (two choices):

```
EVENTLOGRECORD* pEventLogRecord = (EVENTLOGRECORD*) abyBuffer; // OK
EVENTLOGRECORD* pEventLogRecord = (EVENTLOGRECORD*) &abyBuffer[0]; // OK
```

Rule CON#009

Synopsis: Don't compare an address to null
Language: C++
Level: 2
Category: Conversions

Description

Comparing an address to null will always be false. So if this happens it is probably not intended. For example:

```
#include <stdio.h>

class Foo {};

void var(Foo* foo)
{
  if (&foo == nullptr) return;
  printf("YES\n");
}
```

will always print "YES". Probably

```
  if (foo == nullptr) return;
```

was meant.

Error Handling

This chapter concerns error and handling through return values, status codes and C++ exceptions.

Rules

ERR#001	Do not let destructors throw exceptions
ERR#003	Before letting any exceptions propagate out of a member function, make certain that the class invariant holds and, if possible, leave the state of the object unchanged
ERR#005	Check for all errors reported from functions
ERR#006	Don't use exception specifications, but do use noexcept when applicable
ERR#012	Throw only objects of class type
ERR#014	Do not catch objects by value
ERR#015	Always catch exceptions the caller is not supposed to know about
ERR#016	Do not catch exceptions you are not supposed to know about
ERR#017	A catch-all clause must do a rethrow

Rule ERR#001

Synopsis: Do not let destructors throw exceptions
Language: C++
Level: 1
Category: Error Handling

Description

In contrast to what is said in ISC++, do not rely on the std::uncaught_exception function. That function is not always properly implemented by mainstream compilers. Furthermore, there are certain situations where this function is actually useless for the purpose it is intended for!

Any destructor can be called as a result of stack-unwinding due to an exception. If that destructor would throw another exception by itself, then the program will terminate without further cleanup handling. That is almost never a desirable situation. Therefor, destructors shall never throw an exception.

Note that a function called by a destructor *is* allowed to throw an exception! As long as you make sure that the destructor handles any exception if that might be thrown by its implementation.

Rule ERR#003

Synopsis: Before letting any exceptions propagate out of a member function, make certain that the class invariant holds and, if possible, leave the state of the object unchanged
Language: C++
Level: 1
Category: Error Handling

Description

This avoids that the object throwing an exception is left in an inconsistent (undefined) state. For more details see [Abrahams].

Rule ERR#005

Synopsis: Check for all errors reported from functions
Language: C++
Level: 6
Category: Error Handling

Description

If an error reported this way is ignored, there is no easy way of knowing what eventually made the program crash. It seems natural to check status values returned from functions, but in reality there are huge amounts of code written that does not do these checks. The fact that status values can be ignored by the programmer is one of the reasons to why exception handling in most cases is a better way of reporting errors.

Example:

```
// create socket
int socketfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (socketfd < 0) // check status value
{
   // ...
}
```

Using (void) to cast-away the return value is allowed.

```
(void) FunctionWithReturnValue();
```

Rule ERR#006

Synopsis: Don't use exception specifications, but do use noexcept when applicable
Language: C++
Level: 1
Category: Error Handling

Description

Exception specifications are a way to state what kind of exceptions are thrown by a function, e.g.

```
int Gunc() throw(); // will throw nothing
int Hunc() throw(A,B); // can only throw A or B
```

There are a couple of disadvantages with this approach:

- Exception specifications are not checked compile-time. If a function appears to throw another exception as specified in the exception specification then a run-time error will occur that can't be

caught.
- Exception specifications are only allowed for plain functions. It won't work for typedefs for instance.
- Exception specifications result in run-time overhead, thus affecting performance

This is the reason why exception specifications have been deprecated in C++11. The keyword "noexcept(boolean)" has been introduced instead to indicate that a function might throw or won't throw an exception, depending on the value of the boolean, e.g.

```
int Gunc() noexcept(true); // will throw nothing
int Gunc() noexcept; // same as previous one
int Hunc() noexcept(false); // may throw an exception
```

Note that this standard doesn't enforce the use of "noexcept", it only deprecates the use of exception specifications.

Rule ERR#012

Synopsis: Throw only objects of class type
Language: C++
Level: 1
Category: Error Handling

Description

Exceptions pass information up the call stack to a point where error handling can be performed. Class types can have member data with information about the cause of the error, and also the class type itself is further documentation of the cause. User exception types should derive from std::exception or one of its derived classes.

Wrong examples:

```
void m(int x) {
  throw 23; // wrong
  throw x; // wrong

  try {}
  catch (int y) {
    throw; // wrong
    throw y; // wrong
  }
```

Correct examples:

```
void m(int x) {
  try {}
  catch (Exception& e) {
    throw; // right
    throw e; // right
  }
}
```

Rule ERR#014

Synopsis: Do not catch objects by value
Language: C++
Level: 1
Category: Error Handling

Description

If objects are caught by value, there is a risk the exception will be sliced. See for instance the following example:

```
class Exception{};
class SpecialException : public Exception {};

try {
    throw SpecialException();
} catch (Exception e) {
    cout
```

A SpecialException is thrown but its parent Exception is caught. We are catching by value, which results in copying the exception. But since we are catching Exception, not SpecialException, Exception's copy construcor is called, and we lose the SpecialException part of the object.

The right way to write this would be to use a const reference:

```
class Exception{};
class SpecialException : public Exception {};

try {
    throw SpecialException();
} catch (const Exception& e) {
    cout
```

This will result in a reference to the original SpecialException.

Rule ERR#015

Synopsis: Always catch exceptions the caller is not supposed to know about
Language: C++
Level: 9
Category: Error Handling

Description

Hidden implementation details is an important property of well written programs, since it gives you the possibility to make changes without affecting the user.

Imagine a hierarchy of libraries where some libraries are implemented on top of other libraries. To be able to change or replace lower level classes without affecting the user, you must catch all exceptions that the user is not supposed to know about. Otherwise an exception of a class unknown to the user could terminate the program or be caught by a handler with a ... parameter list. In either case, nothing can be said about what caused the exception to be thrown. All exceptions that reach the user should be known to the user, since that will make it possible to explain why the exception was thrown and how to prevent it from being

thrown. You should try to avoid writing programs that simply crash without any proper indication of what went wrong.

Rule ERR#016

Synopsis: Do not catch exceptions you are not supposed to know about
Language: C++
Level: 9
Category: Error Handling

Description

Catch an exception only when you actually can do something about the problem that caused it or at least leave the application in a consistent state. Else throw it along and let someone up the call stack handle it.

Rule ERR#017

Synopsis: A catch-all clause must do a rethrow
Language: C++
Level: 3
Category: Error Handling

Description

If a catch clause catches all possible exceptions without rethrowing, all exceptions are lost.

```
void f()
{
   try
   {
      // Do something
   }
   catch (...)            // catch any exception
   {
      // Cleanup

      throw;              // Always rethrow the exception if catched all
   }
}
```

Only the "main" function of a thread must catch and report all exceptions without rethrowing.

General

This chapter concerns general rules that are not related to a specific part of C++.

Rules

GEN#002	Company C++ code shall be clearly separated from 3rd party code

Rule GEN#002

Synopsis: Company C++ code shall be clearly separated from 3rd party code
Language: C++
Level: 9
Category: General

Description

Maintain a clear separation between in-house developed code and 3rd party source files, which includes generated code. Note that this may require explicit attention in the design phase. In general, interfaces and interoperability with 3rd party code shall be clearly documented in the design.

It is allowed to use interfaces defined by header files of 3rd party libraries directly in company C++ code. An example is the use of standard C++ libraries. It is not obligatory to hide such an interface by means of a company specific C++ wrapper, although there may be good reasons, such as improved portability, to do so in specific cases.

If possible, C++ code that is maintained by the company, should not be located 'embedded' in generated 3rd party code. In any case, files containing such mixed code should be clearly separated from regular company C++ files. Such mixed files will almost certainly contain several coding standard violations, and the relevant design document shall, identify the violated rules and specify alternative rules that do apply. The documentation should also specify which design patterns are used to separate and localize the parts of the implementation that are partly generated and therefore contains coding standard violations.

Naming

This chapter concerns the naming of abstractions used in the code. Names are important for readability and maintainability of the code. By applying naming conventions, code is more comprehensible for others, be it now during development or at a later stage during maintenance. In general, this means that the maintainability and quality of the code increases.

In this chapter, it is important to distinguish between identifiers and names. The name is that part of an identifier that shows its meaning. An identifier consists of a prefix, a name, and a suffix (in that order). The prefix and the suffix are optional. A suffix is only used by tools that generate C++ code, to avoid name collisions with user-written C++ code and is not given further consideration here.

Rules

NAM#002	Do not use identifiers which begin with an underscore ('_') followed by a capital
NAM#008	Do not use identifiers that contain two or more underscores in a row

Rule NAM#002

Synopsis: Do not use identifiers which begin with an underscore ('_') followed by a capital
Language: C++
Level: 1
Category: Naming

Description

Identifier names which begin with an underscore followed by a capital are reserved for compiler specific identifiers.

Rule NAM#008

Synopsis: Do not use identifiers that contain two or more underscores in a row
Language: C++
Level: 1
Category: Naming

Description

Identifier names which contain two or more underscores in a row are reserved for compiler specific identifiers.

Exception: extern variables are excluded from this rule.

Object Allocation

This chapter concerns the allocation and deallocation of objects using the new and delete operators.

Rules

OAL#002	If you overload operator delete, it shall be prepared to accept a null pointer
OAL#003	If you overload operator new for a class, you should have a corresponding operator delete
OAL#004	In case allocated memory has to be deallocated later by someone else, use a unique_ptr with move semantics
OAL#009	Do not overload the global operator new or the global operator delete
OAL#011	Use smart pointers for memory management
OAL#012	Don't use auto_ptr, use unique_ptr instead
OAL#013	Use std::make_shared instead of constructing a shared_ptr from a raw pointer
OAL#018	Don't forget to give scoped variables a name

Rule OAL#002

Synopsis: If you overload operator delete, it shall be prepared to accept a null pointer
Language: C++
Level: 1
Category: Object Allocation

Description

Passing a null pointer to operator delete shall always be allowed, and have no effect. This also applies to operator delete[].

Rule OAL#003

Synopsis: If you overload operator new for a class, you should have a corresponding operator delete
Language: C++
Level: 1
Category: Object Allocation

Description

This also applies to operator new[] and operator delete[].

Rule OAL#004

Synopsis: In case allocated memory has to be deallocated later by someone else, use a unique_ptr with move semantics
Language: C++
Level: 1

Category: Object Allocation

Description

Example:

```
std::unique_ptr<Foo> createFoo() {
  std::unique_ptr<Foo> ptr(new Foo);
  // return unique pointer to Foo
  return ptr;
}

void m() {
  auto foo = createFoo();
  // transfer ownership of foo to bar
  Bar bar(std::move(foo));
}
```

Rule OAL#009

Synopsis: Do not overload the global operator new or the global operator delete
Language: C++
Level: 2
Category: Object Allocation

Description

Other users may depend on the default behavior of these operators. See section 15.6 in ref. [Stroustrup]. If necessary, overload these operators for a specific class. See also Rec. [OAL#003].

Rule OAL#011

Synopsis: Use smart pointers for memory management
Language: C++
Level: 2
Category: Object Allocation

Description

Smart pointers are pointers that free their allocated memory automatically when they go out of scope. In this way, memory leaks are avoided automatically. Use make_shared and make_unique to create smart pointers.

Example (wrong):

```
void m() {
  MyObject* ptr = new MyObject();
  ptr->DoSomething(); // Use the object in some way
  delete ptr; // Destroy the object. Done with it.
}
```

Example (wrong):

```
void m() {
  std::unique_ptr<MyObject> ptr(new MyObject());
  ptr->DoSomething();
}
```

Example (correct):

```
void m() {
  auto ptr = std::make_unique<MyObject>();
  ptr->DoSomething();
}
```

Rule OAL#012

Synopsis: Don't use auto_ptr, use unique_ptr instead
Language: C++
Level: 2
Category: Object Allocation

Description

auto_ptr is deprecated as of C++11. Instead, unique_ptr should be used. It has similar functionality, but improved security (no fake copy assignments), added features (deleters) and support for arrays.

Rule OAL#013

Synopsis: Use std::make_shared instead of constructing a shared_ptr from a raw pointer
Language: C++
Level: 2
Category: Object Allocation

Description

There are a couple of advantages to prefer std::make_shared to the ordinary way of creating a shared pointer.

- Better readibility (see example below)
- Less memory consumption
- Better exception-safety

Example (wrong):

```
std::shared_ptr<Object> ptr(new Object("foo"));
```

Example (correct):

```
std::shared_ptr<Object> ptr = std::make_shared<Object>("foo");
```

Example (correct, even better):

```
auto ptr = std::make_shared<Object>("foo");
```

Rule OAL#018

Synopsis: Don't forget to give scoped variables a name
Language: C++
Level: 1
Category: Object Allocation

Description

If you don't give scoped variables a name, the corresponding anonymous object will be created and destroyed immediately, but not at the end of its scope (as it would be expected). This is extremely dangerous in RAII (Resource Allocation Is Initialization) patterns or SBRM (Scope-Bound Resource Management) patterns, where resource allocation and cleanup depends on object life time, e.g. when allocating heap memory, sockets, files, locks, etc.

Take a look at the following code:

```
void EventListenerHelper::ClearEvents()
{
    boost::mutex::scoped_lock(m_mutex);
    m_events.clear();
}
```

The field "m_mutex" is locked and released immediately after that, so that the clear() operation is actually not locked as expected.

It should have been something like

```
void EventListenerHelper::ClearEvents()
{
    boost::mutex::scoped_lock my_lock(m_mutex);
    m_events.clear();
}
```

instead. The scoped lock is now named "my_lock" and will remain alive until the end of the function scope.

Object Life Cycle

This chapter concerns the declaring, initializing and copying of objects. These guidelines increase robustness and readability of the code. Furthermore, they may also improve the performance.

Rules

OLC#001	If objects of a class should never be copied, then the copy constructor and the copy assignment operator shall be declared as deleted functions
OLC#003	A function must never return, or in any other way give access to, references or pointers to local variables outside the scope in which they are declared
OLC#004	Every variable of a built-in type that is declared is to be given a value before it is used
OLC#005	Don't call virtual functions in constructors and destructors
OLC#006	Variables and types are to be declared with the smallest possible scope
OLC#009	Literals should be used only on the definition of constants and enumerations
OLC#010	Declare each variable in a separate declaration statement
OLC#012	Do not initialise static variables with other (external) static variables
OLC#016	Do not re-declare a visible name in a nested scope
OLC#017	Initialize all data members of built-in types
OLC#018	Let the order of the initializer list be: first base class constructor(s), then data members in the same order of declaration as in the header file
OLC#019	If passing the "this" pointer to a constructor initializer list, be aware that the object is not yet fully constructed
OLC#020	Don't pass member variables to the base class in the constructor's initializer list
OLC#021	Initialize atomic variables correctly

Rule OLC#001

Synopsis: If objects of a class should never be copied, then the copy constructor and the copy assignment operator shall be declared as deleted functions
Language: C++
Level: 2
Category: Object Life Cycle

Description

Use deleted functions instead of old-fashioned private declarations to prohibit copying.

Wrong example:

```
class SomeClass {
private:
  SomeClass(SomeClass const&);
  SomeClass& operator=(SomeClass const&);
};
```

Right example:

```
class SomeClass {
public:
  SomeClass(SomeClass const&) = delete;
```

```
  SomeClass& operator=(SomeClass const&) = delete;
};
```

Rule OLC#003

Synopsis: A function must never return, or in any other way give access to, references or pointers to local variables outside the scope in which they are declared
Language: C++
Level: 2
Category: Object Life Cycle

Description

If a function returns a reference or a pointer to a local variable, the memory to which it refers will already have been deallocated, when this reference or pointer is used. Note that the scope in this rule refers to the lifetime-scope. It is never allowed that a function gives access to a local automatic stack variable. However, if the lifetime of a local variable exceeds the lifetime of its block, then it may well be appropriate to give access to it from other scopes. This might be the case for static or new allocated objects.

Rule OLC#004

Synopsis: Every variable of a built-in type that is declared is to be given a value before it is used
Language: C++
Level: 1
Category: Object Life Cycle

Description

This value can be assigned in the declaration or later on, as long as a variable is assigned a value before it is used.

Some examples:

```
int number = 10;

if (func(number)) ...
```

or

```
int number;
...
number = 10;
if (func(number)) ...
```

This rule can be checked by most modern compilers.

Rule OLC#005

Synopsis: Don't call virtual functions in constructors and destructors
Language: C++
Level: 1
Category: Object Life Cycle

Description

Reason The function called will be that of the object constructed so far, rather than a possibly overriding function in a derived class. This can be most confusing. Worse, a direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results in undefined behavior.

Example, bad

```
class Base {
public:
    virtual void f() = 0;   // not implemented
    virtual void g();       // implemented with Base version
    virtual void h();       // implemented with Base version
    virtual ~Base();        // implemented with Base version
};

class Derived : public Base {
public:
    void g() override;   // provide Derived implementation
    void h() final;      // provide Derived implementation

    Derived()
    {
        // BAD: attempt to call an unimplemented virtual function
        f();

        // BAD: will call Derived::g, not dispatch further virtually
        g();

        // GOOD: explicitly state intent to call only the visible version
        Derived::g();

        // ok, no qualification needed, h is final
        h();
    }
};
```

Note that calling a specific explicitly qualified function is not a virtual call even if the function is virtual.

Note There is nothing inherently wrong with calling virtual functions from constructors and destructors. The semantics of such calls is type safe. However, experience shows that such calls are rarely needed, easily confuse maintainers, and become a source of errors when used by novices.

Rule OLC#006

Synopsis: Variables and types are to be declared with the smallest possible scope
Language: C++
Level: 9
Category: Object Life Cycle

Description

Not only variables must be declared with the smallest possible scope. This also applies to types, structures, enumerations, constants, classes etc.. For example, if an enumeration is used for a return value of a member function, the enumeration is defined in the scope of the class.

```
class CMyClass
{
public:
    enum MyReturnValue
    {
        MYRETURNVALUE_1,
        MYRETURNVALUE_2,
    };

    MyReturnValue
    Foo();
};
```

Rule OLC#009

Synopsis: Literals should be used only on the definition of constants and enumerations
Language: C++
Level: 7
Category: Object Life Cycle

Description

Avoiding literal values usually enhances maintainability and readability. Uncommon literal numeric values are called "magic numbers". Having 2 or more identical magic numbers in the same scope is dangerous. If one needs to change the magic number there is a risk that one forgets to change the others.

Wrong example:

```
if (mode == 23) {
  throw MyException(23);
}
```

Correct example:

```
const int IncorrectState = 23;

if (mode == IncorrectState) {
  throw MyException(IncorrectState);
}
```

Note that the following numbers are considered to be no magic numbers: 0, powers of 2, powers of 10 and the well known angles: 90, 180, 270 and 360. Another exception concerns const array definitions.

Rule OLC#010

Synopsis: Declare each variable in a separate declaration statement
Language: C++

Level: 9
Category: Object Life Cycle

Description

This way the type of a variable can easily be changed when necessary. It also increases the readability.

Rule OLC#012

Synopsis: Do not initialise static variables with other (external) static variables
Language: C++
Level: 6
Category: Object Life Cycle

Description

The standard states that static variables are zero initialised before their first use. The order of initialization is not defined across compilation units. So when a static variable is being initialized with another (external) static variable it might be initialised with only the zero initialised value instead of the programmer intended initialised value.

Violating example:

```
extern int i;

class A
{
public:
    // error: i might be initialised with 0 or the intended value
    static int j = i;
};
```

Non-violating example:

```
int i = 5;

class A
{
public:
    static int j = i;
};
```

Rule OLC#016

Synopsis: Do not re-declare a visible name in a nested scope
Language: C++
Level: 2
Category: Object Life Cycle

Description

This "shadowing" is almost always done inadvertently and can lead to hard to trace bugs.

Example1:

```
int i = 1;
void f()
{
        int i = 2;
        int j = i;   // Violation
}
```

Exception: In case a method parameter has the same name as a field then the following construction can be used: this.x = x

Example2:

```
int i = 1;
void f(int i)
{
        this.i = i; // No violation
        int j = i; // However, this again is a violation!
}
```

See also [INT#026] for a related issue in the context of derived classes.

Rule OLC#017

Synopsis: Initialize all data members of built-in types
Language: C++
Level: 1
Category: Object Life Cycle

Description

Data members of user-defined types that have a default constructor need not be explicitly initialized. Built-in types have default constructors, but these do nothing. So, take care to properly initialize data-members of built-in types.

Pointers to built-in types and pointers to user defined types must also be initialized, because they behave like built-in types.

Note that data member initialization occurs in the constructor's initialization list, rather than assignments in the constructor's body. Only when it is not possible to properly initialize a data member, one should assign a value in the constructor's body.

Rule OLC#018

Synopsis: Let the order of the initializer list be: first base class constructor(s), then data members in the same order of declaration as in the header file
Language: C++

Level: 1
Category: Object Life Cycle

Description

The order in the initializer list is irrelevant to the execution order of the initializers. Putting initializers for data members and base classes in any other order than their actual initialization order is therefore highly confusing and error-prone. A data member could be accessed before it is initialized if the order in the initializer list is incorrect, see Example1.

Virtual base classes are always initialized first. Then base classes, data members and finally the constructor body for the most derived class is run, see Example2.

Example1:

```
class Foo
{
    public:
        Foo(int a, int b) : A(a), B(b), C(A * B) { }  // C is initialized before B which

    private:
        int A;
        int C;    //C initialization depends on A and B and therefore should be declared
        int B;
};
```

Example2:

```
class Derived : public Base  // Base is number 1
{
   public:
      explicit Derived(int i);
      Derived();
   private:
      int  jM;              // jM is number 2
      Base bM;         // bM is number 3
};

Derived::Derived(int i) : Base(i), jM(i), bM(i)
// Recommended order       1        2      3
{
   // Empty
}
```

Rule OLC#019

Synopsis: If passing the "this" pointer to a constructor initializer list, be aware that the object is not yet fully constructed
Language: C++
Level: 5
Category: Object Life Cycle

Description

The "this" pointer can be used in the constructors list of initializers for base classes and member variables. While doing so is valid, you must ensure that those base classes and member variables do not use the passed in "this" pointer to access parts of this object until those parts have been initialized sufficiently. In

particular, any virtual functions should not be called until the rest of the initializer list and the most necessary parts of the constructor code have been executed, nor should any members of *this be accessed until those particular members have been initialized.

Example (wrong):

```
class CC;

class B {
    int &r_x1;
  public:
    B(CC *c);
};

class CC {
  public:
    B b;
    int &r_x2;
    CC(int &i): r_x2(i), b(this) {} // Violation!
};

B::B(CC *c): r_x1( c->r_x2 ) {
}
```

This will crash because b is initialized before r_x2, because the B constructor assumes that r_x2 can be safely duplicated into r_x1, neither of which is true until after r_x2 is initialized.

Example (right):

```
class A {
   int m_x;
   class A_xWrap {
      A& parent;
     public:
      A_xWrap(A* pParent): parent(*pParent) {}
      A_xWrap & operator = (int x) { parent.x_Put(x); return *this; }
      int operator int (void) const { return parent.x_Get(); }
   }
  public:
   A_xWrap x;
   A() : x(this), m_x(2) { } // OK
   int x_Get(void) { return m_x; }
   void x_Put(int i) { if (i>= 0) x = i; }
};
```

This is safe because A_xWrap does not access this until something after this constructor returns tries to use the x field as if it was the m_x field.

Rule OLC#020

Synopsis: Don't pass member variables to the base class in the constructor's initializer list
Language: C++
Level: 2
Category: Object Life Cycle

Description

Because the base class is created before the member variables, these members are not created/initialized yet in the constructor of the base class. If the constructor of the base class uses one of the parameters, an access violation error is inevitable when this constructor is called.

Example:

```
class Base
{
public:
    Base(DMember& m)
    {
        m.Test();      // Access violation!!!
    }
};

class Derived: public Base
{
public:
    Derived()
    : Base(m_Member)
    {
    }
private:
    DMember m_Member;   // Class DMember contains a Test method which uses DMember data
}
```

If an object of Derived is created, then first Base is created with the not-constructed m_Member as input argument. When the Base constructor calls the Test method, an access violation exception will occur.

Rule OLC#021

Synopsis: Initialize atomic variables correctly
Language: C++
Level: 1
Category: Object Life Cycle

Description

If an atomic variable is not default constructed, its behavior will be undefined. So it is important to make sure that atomic variables are initialized in the class definition or via the initializer list of the constructor. For example:

```
class C
{
public:
       C(int x);
private:
       std::atomic<int> i { 0 };
};
```

or alternatively

```
C::C(int x)
    : i(x)
{}
```

Never use atomic_init, because if it is already initialized its behavior is undefined:

```
C::C(int x)
{
    … Some calculation …
    std::atomic_init(&i, y);
}
```

Object Oriented Programming

This chapter concerns parts of object-oriented programming, such as encapsulation, inheritance, polymorphism and associations.

Rules

OOP#001	A class that manages resources shall declare a copy constructor, a copy assignment operator, and a destructor
OOP#002	A public method must never return a non-const reference or pointer to member data
OOP#003	A public method must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects
OOP#004	If you derive from more than one base class with the same parent, that parent shall be a virtual base class that has a default constructor and no data members
OOP#007	Selection statements (if-else and switch) should be used when the control flow depends on an object's value; dynamic binding should be used when the control flow depends on the object's type
OOP#009	Avoid inheritance for parts-of relations
OOP#011	Never redefine an inherited non-virtual method
OOP#013	A base class destructor should be either public and virtual, or protected and non-virtual
OOP#017	It shall be possible to use a pointer or reference to an object of a derived class wherever a pointer or reference to a public base class object is used
OOP#018	When overriding a (virtual) function from a base class, the derived class should give the same const modifier to the function.

Rule OOP#001

Synopsis: A class that manages resources shall declare a copy constructor, a copy assignment operator, and a destructor
Language: C++
Level: 2
Category: Object Oriented Programming

Description

When a copy constructor and a assignment operator do not exist, they are automatically generated by the compiler. This may easily result in runtime errors when an object is copied while this was not intended by the class implementation.

If instances of a class are not intended to be copied, the copy constructor and assignment operator must be declared as deleted functions:

```
class X {
    X& operator=(const X&) = delete;  // Disallow copying
    X(const X&) = delete;
};
```

If instances of a class are intended to be copied and that class has data-members that refer to resources (e.g. pointer to memory) the class must declare and define a copy constructor and an assignment operator.

The copy constructor and assignment operator are declared and defined to make sure that it is not the reference to the resource that is being copied but the data referenced. See also [Meyers] item 11.

Rule OOP#002

Synopsis: A public method must never return a non-const reference or pointer to member data
Language: C++
Level: 2
Category: Object Oriented Programming

Description

This prevents the calling method from being able to manipulate the data member.

Example:

```
class Account
 {
   public:
     Account( int myMoney ) : moneyAmount( myMoney ) {};
     const int& getSafeMoney() const { return moneyAmount; }
     int& getRiskyMoney() const { return moneyAmount; }   // No!
     // ...
   private:
     int moneyAmount;
 };

 Account myAcc(10);   // I'm a poor lonesome programmer a long way from home

 myAcc.getSafeMoney() += 1000000;  // Compilation error: assignment to constant

 myAcc.getRiskyMoney() += 1000000; // myAcc::moneyAmount = 1000010 !!
```

Singleton patterns are an exception to this rule. Well-known examples of other exceptions to this rule are container classes (e.g. std::vector), operator<<, operator= and various Qt methods.

Rule OOP#003

Synopsis: A public method must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects
Language: C++
Level: 2
Category: Object Oriented Programming

Description

Although, the data exists outside the object it is still part of the state of the object. The calling method should not be able to manipulate this data.

Rule OOP#004

Synopsis: If you derive from more than one base class with the same parent, that parent shall be a virtual base class that has a default constructor and no data members
Language: C++
Level: 9
Category: Object Oriented Programming

Description

This rule prevents potential problems caused by multiple assignments by the multiple derived classes of the common base class. It also avoids that the most derived class must explicitly initialize the common base class, which need not be its direct parent.

Rule OOP#007

Synopsis: Selection statements (if-else and switch) should be used when the control flow depends on an object's value; dynamic binding should be used when the control flow depends on the object's type
Language: C++
Level: 9
Category: Object Oriented Programming

Description

Heavy use of the selection statements if / else and switch might be an indication of a poor design. Selection statements should mostly be used when the flow of control depends on the value of an object.

Selection statements are not the best choice if the flow of control depends on the type of an object. If you want to have an extensible set of types that you operate upon, code that uses objects of different types will be difficult and costly to maintain. Each time you need to add a new type, each selection statement must be updated with a new branch. It is best to localize selection statements to a few places in the code. This however requires that you use inheritance and virtual member functions.

Suppose you have a class that is a public base class. It is possible to operate on objects of derived classes without knowing their type if you only call virtual member functions. Such member function calls are dynamically bound, i.e. the function to call is chosen at run-time. Dynamic binding is an essential component of object-oriented programming and we cannot overemphasize the importance that you understand this part of C++. You should try to use dynamic binding instead of selection statements as much as possible. It gives you a more flexible design since you can add classes without rewriting code that only depends on the base class interface.

Rule OOP#009

Synopsis: Avoid inheritance for parts-of relations
Language: C++
Level: 9
Category: Object Oriented Programming

Description

A common mistake is to use *multiple inheritance* for *parts-of* relations (when an object consists of several other objects, these are inherited instead of using instance variables). This can result in strange class hierarchies and less flexible code. In C++ there may be an arbitrary number of instances of a given type; if inheritance is used, direct inheritance from a class may only be used once.

Rule OOP#011

Synopsis: Never redefine an inherited non-virtual method
Language: C++
Level: 2
Category: Object Oriented Programming

Description

In the following code a non-virtual method is redefined, which is not allowed.

```
class Base
{
public: // Methods
    void Foo();
};

class Derived : public Base
{
public: // Methods
    void Foo(); // Not allowed !!!
};
```

The reasons can be found in "Effective C++" item 37 (see [Meyers]). When an inherited non-virtual method may not be redefined, it may also not be made virtual in the derived class, as in:

```
class Base
{
public: // Methods
    void Foo();
};

class Derived : public Base
{
public: // Methods
    virtual void Foo(); // Not allowed !!!
};
```

Exception: this rule doesn't hold for classes that have been derived privately from its base class because in such a case no harm can be done.

Rule OOP#013

Synopsis: A base class destructor should be either public and virtual, or protected and non-virtual
Language: C++
Level: 1

Category: Object Oriented Programming

Description

Reason To prevent undefined behavior. If the destructor is public, then calling code can attempt to destroy a derived class object through a base class pointer, and the result is undefined if the base class's destructor is non-virtual. If the destructor is protected, then calling code cannot destroy through a base class pointer and the destructor does not need to be virtual; it does need to be protected, not private, so that derived destructors can invoke it. In general, the writer of a base class does not know the appropriate action to be done upon destruction.

Example, bad

```
struct Base {  // BAD: implicitly has a public non-virtual destructor
    virtual void f();
};

struct D : Base {
    string s {"a resource needing cleanup"};
    ~D() { /* ... do some cleanup ... */ }
    // ...
};

void use()
{
    unique_ptr p = make_unique();
    // ...
} // p's destruction calls ~Base(), not ~D(), which leaks D::s and possibly more
```

Note A virtual function defines an interface to derived classes that can be used without looking at the derived classes. If the interface allows destroying, it should be safe to do so.

Note A destructor must be non-private or it will prevent using the type:

```
class X {
    ~X();   // private destructor
    // ...
};

void use()
{
    X a;                        // error: cannot destroy
    auto p = make_unique();  // error: cannot destroy
}
```

Exception We can imagine one case where you could want a protected virtual destructor: When an object of a derived type (and only of such a type) should be allowed to destroy another object (not itself) through a pointer to base. We haven't seen such a case in practice, though.

Rule OOP#017

Synopsis: It shall be possible to use a pointer or reference to an object of a derived class wherever a pointer or reference to a public base class object is used
Language: C++
Level: 3
Category: Object Oriented Programming

Description

A class inherits from another class to reuse either the implementation or the class interface. Public inheritance makes it possible to write code that only depends on the base class interface, not the implementation. Public inheritance should only be used if derived class objects are supposed to be operated upon through base class pointers or references.

This rule is known as the "Liskov Substitution Principle", see [Liskov] for more details.

Rule OOP#018

Synopsis: When overriding a (virtual) function from a base class, the derived class should give the same const modifier to the function.
Language: C++
Level: 2
Category: Object Oriented Programming

Description

This prevents unintentionally calling the function of the base class.

```
class Base
{
public: // Methods
    virtual void f() const;
};

class Derived : public Base
{
public: // Methods
    virtual void f(); //This is not an override, not allowed !!!
};
```

The reason is when a client attempts a polymorphic call to f() through a const pointer or const reference to Base, it will call Base::f() and not Derived::f(), which is syntactically correct but probably surprising. The same care should be used when overriding a function with a parameter that is a pointer or reference to const.

Optimization and Performance

This chapter contains rules that optimize code and improve performance.

Rules

OPT#001	Avoid duplicated code and data
OPT#002	Optimize code only if you know that you have a performance problem. Think twice before you begin

Rule OPT#001

Synopsis: Avoid duplicated code and data
Language: C++
Level: 4
Category: Optimization and Performance

Description

When duplicating code and data makes the program unnecessary large, which can effect the overall performance of the system. More time will be spent in swapping programs in and out of memory. Furthermore, the code becomes harder to maintain, because when the code or data has to be changed, it has to be done at multiple source locations.

Rule OPT#002

Synopsis: Optimize code only if you know that you have a performance problem. Think twice before you begin
Language: C++
Level: 7
Category: Optimization and Performance

Description

A lot of time may be lost optimizing code without significant performance improvement. If you have a performance problem, use a performance-analyzing tool, to determine where a bottleneck exists.

Parts of C++ to Avoid

This chapter concerns several parts of C++ which should be avoided. New standard C++ library classes and templates replace in most cases functions inherited from the C standard library. Also, for certain parts of the language that are inherited from C, better language constructs exist or there are classes or templates to use instead.

Rules

PCA#001	Don't use malloc, calloc, realloc, free and cfree for memory management
PCA#002	Do not assume that an enumerator has a specific value
PCA#003	Use overloaded functions and chained function calls instead of functions with an unspecified number of arguments
PCA#005	Use the iostream library instead of C-style I/O
PCA#006	Do not use setjmp and longjmp
PCA#008	Do not redefine keywords
PCA#009	Use an array class instead of built-in arrays
PCA#010	Do not use unions with non-POD types
PCA#011	Do not use bit-fields
PCA#016	Use enum classes instead of old-style enums
PCA#017	Don't compare unrelated enumerations
PCA#018	Use functionality from the std library if possible

Rule PCA#001

Synopsis: Don't use malloc, calloc, realloc, free and cfree for memory management
Language: C++
Level: 1
Category: Parts of C++ to Avoid

Description

In C malloc, realloc and free are used to allocate memory explicitly on the heap. In C++ one should use automated memory management by means of smart pointers instead.

Rule PCA#002

Synopsis: Do not assume that an enumerator has a specific value
Language: C++
Level: 4
Category: Parts of C++ to Avoid

Description

A common pitfall is that it is assumed that the enumerators of an enumeration run from 0 to (max - 1). Although each enumerator has an integer value, no code may be written that relies on the integer value of the enumerator.

Incorrect example:

```
enum Color { red, green, blue };
Color color;
if (color > 1) { ... }
color = 2;
```

Correct example:

```
enum Color { red, green, blue };
Color color;
if (color > green) { ... }
color = blue;
```

Important note. Instead of using old style enums, it is better to use enum classes. See also [PCA#016]. Enum classes don't allow conversions between enum values and integers. The compiler will issue a compiler error for this.

Rule PCA#003

Synopsis: Use overloaded functions and chained function calls instead of functions with an unspecified number of arguments
Language: C++
Level: 4
Category: Parts of C++ to Avoid

Description

Functions or methods using variable arguments should not be used unless very good reasons exist to do so. This is not advised since the strong type checking provided by C++ is thereby avoided. The intended effect can often be reached using function overloading.

Rule PCA#005

Synopsis: Use the iostream library instead of C-style I/O
Language: C++
Level: 9
Category: Parts of C++ to Avoid

Description

The iostream library is more type-safe, efficient, and extensible than stdio. See also [GEN#001].

Rule PCA#006

Synopsis: Do not use setjmp and longjmp
Language: C++
Level: 1

Category: Parts of C++ to Avoid

Rule PCA#008

Synopsis: Do not redefine keywords
Language: C++
Level: 1
Category: Parts of C++ to Avoid

Description

Do not use the preprocessor to redefine C++ keywords. Abominations such as #define private public are most certainly forbidden, but also the VC++ generated code #define new DEBUG_NEW must be removed.

Rule PCA#009

Synopsis: Use an array class instead of built-in arrays
Language: C++
Level: 9
Category: Parts of C++ to Avoid

Description

Especially the std::vector class template is usually a better alternative than built-in arrays for several reasons:

- They allocate memory from the free space when increasing in size.
- They are NOT a pointer in disguise.
- They can increase/decrease in size run-time.
- They can do range checking using at().
- A vector knows its size, so you don't have to count elements.

The most compelling reason to use a vector is that it frees you from explicit memory management, and it does not leak memory. A vector keeps track of the memory it uses to store its elements. When a vector needs more memory for elements, it allocates more; when a vector goes out of scope, it frees that memory. Therefore, the user needs not be concerned with the allocation and deallocation of memory for vector elements.

An exception to this rule are built-in arrays that are used in a local scope.

Rule PCA#010

Synopsis: Do not use unions with non-POD types
Language: C++
Level: 6
Category: Parts of C++ to Avoid

Description

C++11 gave us the possibility to use non-POD types within unions. E.g.

```
union
{
    T one;
    V two;
} foo;
```

If you use these kind of unions you are mostly on your own, e.g. concerning deletion of union objects and how to determine what the default value will be. A note in the C++ standard explains this (9.5/2): "If any non-static data member of a union has a non-trivial default constructor (12.1), copy constructor (12.8), move constructor (12.8), copy assignment operator (12.8), move assignment operator (12.8), or destructor (12.4), the corresponding member function of the union must be user-provided or it will be implicitly deleted (8.4.3) for the union."

Rule PCA#011

Synopsis: Do not use bit-fields
Language: C++
Level: 5
Category: Parts of C++ to Avoid

Description

If breaking this rule is necessary, one must be very well aware of the undefined properties of bit-fields, and provide adequate documentation. In general, prevent to rely on the exact memory layout of objects: see also [POR#020]. When the purpose is to manipulate a set of a number of bits, then the std::bitset class template is a better alternative.

Rule PCA#016

Synopsis: Use enum classes instead of old-style enums
Language: C++
Level: 7
Category: Parts of C++ to Avoid

Description

Old-style C++ enums are essentially integers; they could be compared with integers or with other enums of different types without getting compilation errors. Most of the times this is not intentional. Now with strongly typed enums, the compiler will not accept this any more. If really needed, you can always use a typecast.

Another limitation of old-style enum values is that they are unscoped--in other words, you couldn't have two enumerations that shared the same name:

```
// this code won't compile because both enums contain BLUE!
enum Color {RED, GREEN, BLUE};
enum Feelings {EXCITED, MOODY, BLUE};
```

This in contrast with enum classes:

```
// this code will compile (if your compiler supports C++11 strongly typed enums)
enum class Color {RED, GREEN, BLUE};
enum class Feelings {EXCITED, MOODY, BLUE};
```

The use of the word class is meant to indicate that each enum type really is different and not comparable to other enum types. Strongly typed enums, enum classes, also have better scoping. Each enum value is scoped within the name of the enum class. In other words, to access the enum values, you must write:

```
Color color = Color::GREEN;
if ( Color::RED == color )
{
    // the color is red
}
```

Yet another advantage of enum classes is that it is possible to specify the underlying type (that is: the byte size), which is very useful when using this enum declaration in definition of network protocols or other binary data. Otherwise the size of the enum is compiler dependent (usually machine word size), which is not portable. Example:

```
enum class Permissions : unsigned short {Readable = 0x4, Writable = 0x2, Executable = 0x
```

Rule PCA#017

Synopsis: Don't compare unrelated enumerations
Language: C++
Level: 2
Category: Parts of C++ to Avoid

Description

It is possible to compare old style enumerations with each other. Although there is no use to compare such enumerations, the compiler won't warn about this because enumerations are treated as integers.

Wrong example:

```
enum Color { red, yellow, blue };
enum Status { OK, NOK };

void m(Color col) {
  if (col == OK) { // compiles fine, but is not intended
    dosomething();
  }
}
```

Right example:

```
enum Color { red, yellow, blue };
enum Status { OK, NOK };

void m(Color col) {
  if (col == red) {
    dosomething();
  }
}
```

Rule PCA#018

Synopsis: Use functionality from the std library if possible
Language: C++
Level: 5
Category: Parts of C++ to Avoid

Description

It is better to use the standard library std if possible because it minimizes the number of external dependencies. For instance, since the introduction of C++11, a lot of Boost functionality is now available from the standard library std.

Sometimes there is a one to one relation between std and Boost such as boost::bind that is replaced by std::bind. But sometimes this relation is less trivial, e.g. to make a class non-copyable one should derive from the boost::noncopyable class in the Boost case:

```
class X : private boost::noncopyable {};
```

whereas in C++11 this is done differently by using "delete" for the constructor and copy constructor:

```
class X {
  X(const X&) = delete;
  X& operator=(const X&) = delete;
};
```

Preprocessor

This chapter concerns the usage of the preprocessor.

Rules

PRE#001	Do not define macros instead of constants, enums, or type definitions
PRE#002	Use parentheses around macro and macro parameters
PRE#003	Be aware of side-effects when using macros
PRE#004	Do not use the preprocessor directive #define to obtain more efficient code; instead, use inline or template methods/functions

Rule PRE#001

Synopsis: Do not define macros instead of constants, enums, or type definitions
Language: C++
Level: 7
Category: Preprocessor

Description

A disadvantage of macros is that they can have side-effects and that they are not type-safe. In C++ alternatives exists that do not have these drawbacks. An alternative for a macro constant is the use of a const or enum, as in:

```
#define MAX_BUFFER_LENGTH  1000        // Not ok.
const ULONG ulMaxBufferLength = 1000;  // Ok.
enum {MAX_BUFFER_LENGTH = 1000};       // Ok.
```

Rule PRE#002

Synopsis: Use parentheses around macro and macro parameters
Language: C++
Level: 4
Category: Preprocessor

Description

If a macro can be used in an expression or the parameter passed to a macro can be an expression, use parenthesis around them.

Consider a macro that calculates the square of number.

```
// Not allowed !!!
#define SQUARE(p) p * p
```

When the macro is used as follows:

```
SQUARE(5 + 2)
```

it is expanded into:

```
5 + 2 * 5 + 2
```

The result is 17 instead of 49.

To prevent such unexpected results, always use parentheses around the entire macro and macro parameters, as in:

```
// OK
#define SQUARE(p) ((p) * (p))
```

An exception to this rule is when a macro is defined as a single token, as in:

```
// OK
#define SOMETHING SOMETHING_ELSE
```

Rule PRE#003

Synopsis: Be aware of side-effects when using macros
Language: C++
Level: 1
Category: Preprocessor

Description

Assume we have the following macro:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

When the macro is used as follows:

```
MAX(i++, j)
```

it is expanded as follows:

```
i++ > j ? i++ : j
```

If i is greater than j, it is incremented twice.

Rule PRE#004

Synopsis: Do not use the preprocessor directive #define to obtain more efficient code; instead, use inline or template methods/functions
Language: C++
Level: 9
Category: Preprocessor

Description

An advantage of inline or template methods/functions is that they don't have side-effects as macros have (see [PRE#003]). Another advantage is that inline or template methods/functions are type-safe.

A macro "function" can be defined as a template function, as in:

```
#define MAX(p1, p2) (((p1) > (p2)) ? (p1) : (p2))

template<class T> T&
Max(
    const T& p1,
    const T& p2);
```

Security

Security rules from the Software Engineering Institute at the Carnegie Mellon University.

Rules

ARR38-C	Guarantee that library functions do not form invalid pointers
DCL50-CPP	Do not define a C-style variadic function
ENV33-C	Do not call system()
ERR33-C	Detect and handle standard library errors
ERR54-CPP	Catch handlers should order their parameter types from most derived to least derived
EXP34-C	Do not dereference null pointers
EXP53-CPP	Do not read uninitialized memory
EXT01-CPP	The called function is unsafe for security related code
EXT02-CPP	non-constant printf format string may be susceptible to format string attacks
EXT03-CPP	Calling a function which may pose a security risk if it is used inappropriately
EXT04-CPP	Using an insecure temporary file creation function
EXT05-CPP	A user-land pointer is dereferenced without safety checks in the kernel
FIO30-C	Exclude user input from format strings
FIO34-C	Distinguish between characters read from a file and EOF or WEOF
FIO37-C	Do not assume that fgets() or fgetws() returns a nonempty string when successful
FIO45-C	Avoid TOCTOU race conditions while accessing files
MEM50-CPP	Do not access freed memory
MEM56-CPP	Do not store an already-owned pointer value in an unrelated smart pointer
MSC30-C	Do not use the rand() function for generating pseudorandom numbers
MSC33-C	Do not pass invalid data to the asctime() function
MSC51-CPP	Ensure your random number generator is properly seeded
STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator
STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string
STR38-C	Do not confuse narrow and wide character strings and functions
STR50-CPP	Guarantee that storage for strings has sufficient space for character data and the null terminator
STR51-CPP	Do not attempt to create a std::string from a null pointer

Rule ARR38-C

Synopsis: Guarantee that library functions do not form invalid pointers
Language: C++
Level: 1
Category: Security

Description

C library functions that make changes to arrays or objects take at least two arguments: a pointer to the array or object and an integer indicating the number of elements or bytes to be manipulated. For the

purposes of this rule, the element count of a pointer is the size of the object to which it points, expressed by the number of elements that are valid to access. Supplying arguments to such a function might cause the function to form a pointer that does not point into or just past the end of the object, resulting in undefined behavior.

Annex J of the C Standard [ISO/IEC 9899:2011] states that it is undefined behavior if the "pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid." .

In the following code,

```
int arr[ 5 ];
int *p = arr;

unsigned char *p2 = (unsigned char *)arr;
unsigned char *p3 = arr + 2 ;
void *p4 = arr;
```

the element count of the pointer `p` is `sizeof(arr) / sizeof(arr[0])` , that is, `5` . The element count of the pointer `p2` is `sizeof(arr)` , that is, `20` , on implementations where `sizeof(int) == 4` . The element count of the pointer `p3` is `12` on implementations where `sizeof(int) == 4` , because `p3` points two elements past the start of the array `arr` . The element count of `p4` is treated as though it were `unsigned char *` instead of `void *` , so it is the same as `p2` .

Pointer + Integer

The following standard library functions take a pointer argument and a size argument, with the constraint that the pointer must point to a valid memory object of at least the number of elements indicated by the size argument.

`fgets()`	`fgetws()`	`mbstowcs()`[1]	`wcstombs()`[1]
`mbrtoc16()`[2]	`mbrtoc32()`[2]	`mbsrtowcs()`[1]	`wcsrtombs()`[1]
`mbtowc()`[2]	`mbrtowc()`[1]	`mblen()`	`mbrlen()`
`memchr()`	`wmemchr()`	`memset()`	`wmemset()`
`strftime()`	`wcsftime()`	`strxfrm()`[1]	`wcsxfrm()`[1]
`strncat()`[2]	`wcsncat()`[2]	`snprintf()`	`vsnprintf()`
`swprintf()`	`vswprintf()`	`setvbuf()`	`tmpnam_s()`
`snprintf_s()`	`sprintf_s()`	`vsnprintf_s()`	`vsprintf_s()`
`gets_s()`	`getenv_s()`	`wctomb_s()`	`mbstowcs_s()`[3]
`wcstombs_s()`[3]	`memcpy_s()`[3]	`memmove_s()`[3]	`strncpy_s()`[3]
`strncat_s()`[3]	`strtok_s()`[2]	`strerror_s()`	`strnlen_s()`
`asctime_s()`	`ctime_s()`	`snwprintf_s()`	`swprintf_s()`
`vsnwprintf_s()`	`vswprintf_s()`	`wcsncpy_s()`[3]	`wmemcpy_s()`[3]
`wmemmove_s()`[3]	`wcsncat_s()`[3]	`wcstok_s()`[2]	`wcsnlen_s()`
`wcrtomb_s()`	`mbsrtowcs_s()`[3]	`wcsrtombs_s()`[3]	`memset_s()`[4]

[1] Takes two pointers and an integer, but the integer specifies the element count only of the output buffer, not of the input buffer.
[2] Takes two pointers and an integer, but the integer specifies the element count only of the input buffer, not of the output buffer.
[3] Takes two pointers and two integers; each integer corresponds to the element count of one of the pointers.
[4] Takes a pointer and two size-related integers; the first size-related integer parameter specifies the number of bytes available in the buffer; the second size-related integer parameter specifies the number of bytes to

write within the buffer.

For calls that take a pointer and an integer size, the given size should not be greater than the element count of the pointer.

Noncompliant Code Example (Element Count)

In this noncompliant code example, the incorrect element count is used in a call to `wmemcpy()`. The `sizeof` operator returns the size expressed in bytes, but `wmemcpy()` uses an element count based on `wchar_t *`.

```
#include <string.h>
#include <wchar.h>

static const char str[] = "Hello world";
static const wchar_t w_str[] = L"Hello world";
void func( void ) {
  char buffer[ 32 ];
  wchar_t w_buffer[ 32 ];
  memcpy(buffer, str, sizeof(str)); /* Compliant */
  wmemcpy(w_buffer, w_str, sizeof(w_str)); /* Noncompliant */
}
```

Compliant Solution (Element Count)

When using functions that operate on pointed-to regions, programmers must always express the integer size in terms of the element count expected by the function. For example, `memcpy()` expects the element count expressed in terms of `void *`, but `wmemcpy()` expects the element count expressed in terms of `wchar_t *`. Instead of the `sizeof` operator, functions that return the number of elements in the string are called, which matches the expected element count for the copy functions. In the case of this compliant solution, where the argument is an array `A` of type `T`, the expression `sizeof(A) / sizeof(T)`, or equivalently `sizeof(A) / sizeof(*A)`, can be used to compute the number of elements in the array.

```
#include <string.h>
#include <wchar.h>

static const char str[] = "Hello world";
static const wchar_t w_str[] = L"Hello world";
void func( void ) {
  char buffer[ 32 ];
  wchar_t w_buffer[ 32 ];
  memcpy(buffer, str, strlen(str) + 1 );
  wmemcpy(w_buffer, w_str, wcslen(w_str) + 1 );
}
```

Noncompliant Code Example (Pointer + Integer)

This noncompliant code example assigns a value greater than the number of bytes of available memory to `n`, which is then passed to `memset()`:

```
#include <stdlib.h>
#include <string.h>

void f1(size_t nchars) {
```

```
  char *p = (char *)malloc(nchars);
  /* ... */
  const size_t n = nchars + 1;
  /* ... */
  memset(p, 0, n);
}
```

Compliant Solution (Pointer + Integer)

This compliant solution ensures that the value of `n` is not greater than the number of bytes of the dynamic memory pointed to by the pointer `p` :

```
#include <stdlib.h>
#include <string.h>

void f1(size_t nchars) {
  char *p = (char *)malloc(nchars);
  /* ...  */
  const size_t n = nchars;
  /* ...  */
  memset(p, 0, n);
}
```

Noncompliant Code Example (Pointer + Integer)

In this noncompliant code example, the element count of the array `a` is `ARR_SIZE` elements. Because `memset()` expects a byte count, the size of the array is scaled incorrectly by `sizeof(int)` instead of `sizeof(long)` , which can form an invalid pointer on architectures where `sizeof(int) != sizeof(long)` .

```
#include <string.h>

void f2(void) {
  const size_t ARR_SIZE = 4;
  long a[ARR_SIZE];
  const size_t n = sizeof(int) * ARR_SIZE;
  void *p = a;

  memset(p, 0, n);
}
```

Compliant Solution (Pointer + Integer)

In this compliant solution, the element count required by `memset()` is properly calculated without resorting to scaling:

```
#include <string.h>

void f2(void) {
  const size_t ARR_SIZE = 4;
  long a[ARR_SIZE];
  const size_t n = sizeof(a);
  void *p = a;

  memset(p, 0, n);
}
```

Two Pointers + One Integer

The following standard library functions take two pointer arguments and a size argument, with the constraint that both pointers must point to valid memory objects of at least the number of elements indicated by the size argument.

`memcpy()` `wmemcpy()` `memmove()` `wmemmove()`
`strncpy()` `wcsncpy()` `memcmp()` `wmemcmp()`
`strncmp()` `wcsncmp()` `strcpy_s()` `wcscpy_s()`
`strcat_s()` `wcscat_s()`

For calls that take two pointers and an integer size, the given size should not be greater than the element count of either pointer.

Noncompliant Code Example (Two Pointers + One Integer)

In this noncompliant code example, the value of `n` is incorrectly computed, allowing a read past the end of the object referenced by `q` :

```
#include <string.h>

void f4() {
  char p[ 40 ];
  const char *q = "Too short" ;
  size_t n = sizeof(p);
  memcpy(p, q, n);
}
```

Compliant Solution (Two Pointers + One Integer)

This compliant solution ensures that `n` is equal to the size of the character array:

```
#include <string.h>

void f4() {
  char p[ 40 ];
  const char *q = "Too short" ;
  size_t n = sizeof(p) < strlen(q) + 1 ? sizeof(p) : strlen(q) + 1 ;
  memcpy(p, q, n);
}
```

One Pointer + Two Integers

The following standard library functions take a pointer argument and two size arguments, with the constraint that the pointer must point to a valid memory object containing at least as many bytes as the product of the two size arguments.

`bsearch()` `bsearch_s()` `qsort()` `qsort_s()`
`fread()` `fwrite()`

For calls that take a pointer and two integers, one integer represents the number of bytes required for an individual object, and a second integer represents the number of elements in the array. The resulting product of the two integers should not be greater than the element count of the pointer were it expressed as an `unsigned char *`.

Noncompliant Code Example (One Pointer + Two Integers)

This noncompliant code example allocates a variable number of objects of type `struct obj` . The function checks that `num_objs` is small enough to prevent wrapping, in compliance with INT30-C. Ensure that unsigned integer operations do not wrap. The size of `struct obj` is assumed to be 16 bytes to account for padding to achieve the assumed alignment of `long long` . However, the padding typically depends on the target architecture, so this object size may be incorrect, resulting in an incorrect element count.

```
#include <stdint.h>
#include <stdio.h>

struct obj {
  char c;
  long long i;
};

void func(FILE *f, struct obj *objs, size_t num_objs) {
  const size_t obj_size = 16;
  if (num_objs > (SIZE_MAX / obj_size) ||
      num_objs != fwrite(objs, obj_size, num_objs, f)) {
    /* Handle error */
  }
}
```

Compliant Solution (One Pointer + Two Integers)

This compliant solution uses the `sizeof` operator to correctly provide the object size and `num_objs` to provide the element count:

```
#include <stdint.h>
#include <stdio.h>

struct obj {
  char c;
  long long i;
};

void func(FILE *f, struct obj *objs, size_t num_objs) {
  const size_t obj_size = sizeof *objs;
  if (num_objs > (SIZE_MAX / obj_size) ||
      num_objs != fwrite(objs, obj_size, num_objs, f)) {
    /* Handle error */
  }
}
```

Noncompliant Code Example (One Pointer + Two Integers)

In this noncompliant code example, the function `f()` calls `fread()` to read `nitems` of type `wchar_t` , each `size` bytes in size, into an array of `BUFFER_SIZE` elements, `wbuf` . However, the expression used to compute the value of `nitems` fails to account for the fact that, unlike the size of `char` , the size of `wchar_t` may be greater than 1. Consequently, `fread()` could attempt to form pointers past the end of `wbuf` and use them to assign values to nonexistent elements of the array. Such an attempt is undefined behavior. (See undefined behavior 109 .) A likely consequence of this undefined behavior is a buffer overflow. For a discussion of this programming error in the Common Weakness

Enumeration database, see CWE-121, "Stack-based Buffer Overflow," and CWE-805, "Buffer Access with Incorrect Length Value."

```
#include <stddef.h>
#include <stdio.h>

void f( FILE *file) {
  enum { BUFFER_SIZE = 1024 };
  wchar_t wbuf[BUFFER_SIZE];

  const size_t size = sizeof (*wbuf);
  const size_t nitems = sizeof (wbuf);

  size_t nread = fread (wbuf, size, nitems, file);
  /* ... */
}
```

Compliant Solution (One Pointer + Two Integers)

This compliant solution correctly computes the maximum number of items for `fread()` to read from the file:

```
#include <stddef.h>
#include <stdio.h>

void f( FILE *file) {
  enum { BUFFER_SIZE = 1024 };
  wchar_t wbuf[BUFFER_SIZE];

  const size_t size = sizeof (*wbuf);
  const size_t nitems = sizeof (wbuf) / size;

  size_t nread = fread (wbuf, size, nitems, file);
  /* ... */
}
```

Noncompliant Code Example (Heartbleed)

CERT vulnerability 720951 describes a vulnerability in OpenSSL versions 1.0.1 through 1.0.1f, popularly known as "Heartbleed." This vulnerability allows an attacker to steal information that under normal conditions would be protected by Secure Socket Layer/Transport Layer Security (SSL/TLS) encryption.

Despite the seriousness of the vulnerability, Heartbleed is the result of a common programming error and an apparent lack of awareness of secure coding principles. Following is the vulnerable code:

```
int dtls1_process_heartbeat(SSL *s) {
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  pl = p;
```

```
  /* ... More code ... */

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    /*
     * Allocate memory for the response; size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding.
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;

    /* Enter response type, length, and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);

    /* ... More code ... */
  }
  /* ... More code ... */
}
```

This code processes a "heartbeat" packet from a client. As specified in RFC 6520, when the program receives a heartbeat packet, it must echo the packet's data back to the client. In addition to the data, the packet contains a length field that conventionally indicates the number of bytes in the packet data, but there is nothing to prevent a malicious packet from lying about its data length.

The `p` pointer, along with `payload` and `p1` , contains data from a packet. The code allocates a `buffer` sufficient to contain `payload` bytes, with some overhead, then copies `payload` bytes starting at `p1` into this buffer and sends it to the client. Notably absent from this code are any checks that the payload integer variable extracted from the heartbeat packet corresponds to the size of the packet data. Because the client can specify an arbitrary value of `payload` , an attacker can cause the server to read and return the contents of memory beyond the end of the packet data, which violates INT04-C. Enforce limits on integer values originating from tainted sources. The resulting call to `memcpy()` can then copy the contents of memory past the end of the packet data and the packet itself, potentially exposing sensitive data to the attacker. This call to `memcpy()` violates ARR38-C. Guarantee that library functions do not form invalid pointers. A version of ARR38-C also appears in ISO/IEC TS 17961:2013, "Forming invalid pointers by library functions [libptr]." This rule would require a conforming analyzer to diagnose the Heartbleed vulnerability.

Compliant Solution (Heartbleed)

OpenSSL version 1.0.1g contains the following patch, which guarantees that `payload` is within a valid range. The range is limited by the size of the input record.

```
int dtls1_process_heartbeat(SSL *s) {
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
```

```
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* ... More code ... */

  /* Read type and payload length first */
  if (1 + 2 + 16 > s->s3->rrec.length)
    return 0; /* Silently discard */
  hbtype = *p++;
  n2s(p, payload);
  if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* Silently discard per RFC 6520 */
  pl = p;

  /* ... More code ... */

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    /*
     * Allocate memory for the response; size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding.
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;
    /* Enter response type, length, and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    /* ... More code ... */
  }
  /* ... More code ... */
}
```

Rule DCL50-CPP

Synopsis: Do not define a C-style variadic function
Language: C++
Level: 1
Category: Security

Description

Functions can be defined to accept more formal arguments at the call site than are specified by the parameter declaration clause. Such functions are called *variadic* functions because they can accept a variable number of arguments from a caller. C++ provides two mechanisms by which a variadic function can be defined: function parameter packs and use of a C-style ellipsis as the final parameter declaration.

Variadic functions are flexible because they accept a varying number of arguments of differing types. However, they can also be hazardous. A variadic function using a C-style ellipsis (hereafter called a *C-style variadic function*) has no mechanisms to check the type safety of arguments being passed to the

function or to check that the number of arguments being passed matches the semantics of the function definition. Consequently, a runtime call to a C-style variadic function that passes inappropriate arguments yields undefined behavior. Such undefined behavior could be exploited to run arbitrary code.

Do not define C-style variadic functions. (The declaration of a C-style variadic function that is never defined is permitted, as it is not harmful and can be useful in unevaluated contexts.)

Issues with C-style variadic functions can be avoided by using variadic functions defined with function parameter packs for situations in which a variable number of arguments should be passed to a function. Additionally, function currying can be used as a replacement to variadic functions. For example, in contrast to C's `printf()` family of functions, C++ output is implemented with the overloaded single-argument `std::cout::operator<<()` operators.

Noncompliant Code Example

This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value `0` is found. Calling this function without passing the value `0` as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an `int` also results in undefined behavior.

```
#include <cstdarg>

int add( int first, int second, ...) {
   int r = first + second;
   va_list va;
   va_start (va, second);
   while ( int v = va_arg (va, int )) {
     r += v;
   }
   va_end (va);
   return r;
}
```

Compliant Solution (Recursive Pack Expansion)

In this compliant solution, a variadic function using a function parameter pack is used to implement the `add()` function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with `0` . Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior.

```
#include <type_traits>

template < typename Arg, typename
std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Arg s) { return f + s; }

template < typename Arg, typename ... Ts, typename
std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Ts... rest) {
   return f + add(rest...);
}
```

This compliant solution makes use of `std::enable_if` to ensure that any nonintegral argument value results in an ill-formed program.

Compliant Solution (Braced Initializer List Expansion)

An alternative compliant solution that does not require recursive expansion of the function parameter pack instead expands the function parameter pack into a list of values as part of a braced initializer list. Since narrowing conversions are not allowed in a braced initializer list, the type safety is preserved despite the `std::enable_if` not involving any of the variadic arguments.

```
#include <type_traits>

template < typename Arg, typename ... Ts, typename
std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg i, Arg j, Ts... all) {
  int values[] = { j, all... };
  int r = i;
  for (auto v : values) {
    r += v;
  }
  return r;
}
```

Exceptions

DCL50-CPP-EX1: It is permissible to define a C-style variadic function if that function also has external C language linkage. For instance, the function may be a definition used in a C library API that is implemented in C++.

DCL50-CPP-EX2: As stated in the normative text, C-style variadic functions that are declared but never defined are permitted. For example, when a function call expression appears in an unevaluated context, such as the argument in a `sizeof` expression, overload resolution is performed to determine the result type of the call but does not require a function definition. Some template metaprogramming techniques that employ SFINAE use variadic function declarations to implement compile-time type queries, as in the following example.

```
template < typename Ty>
class has_foo_function {
  typedef char yes[1];
  typedef char no[2];

  template < typename Inner>
  static yes& test(Inner *I, decltype(I->foo()) * = nullptr); //
Function is never defined.

  template < typename >
  static no& test(...); // Function is never defined.

public :
  static const bool value = sizeof (test<Ty>(nullptr)) == sizeof (yes);
};
```

In this example, the value of `value` is determined on the basis of which overload of `test()` is selected. The declaration of `Inner *I` allows use of the variable `I` within the `decltype` specifier, which results in a pointer of some (possibly `void`) type, with a default value of `nullptr`. However, if there is no declaration of `Inner::foo()`, the `decltype` specifier will be ill-formed, and that variant of `test()` will not be a candidate function for overload resolution due to SFINAE. The result is that the C-style variadic function variant of `test()` will be the only function in the candidate set. Both `test()`

functions are declared but never defined because their definitions are not required for use within an unevaluated expression context.

Rule ENV33-C

Synopsis: Do not call system()
Language: C++
Level: 1
Category: Security

Description

The C Standard `system()` function executes a specified command by invoking an implementation-defined command processor, such as a UNIX shell or `CMD.EXE` in Microsoft Windows. The POSIX `popen()` and Windows `_popen()` functions also invoke a command processor but create a pipe between the calling program and the executed command, returning a pointer to a stream that can be used to either read from or write to the pipe [IEEE Std 1003.1:2013].

Use of the system() function can result in exploitable vulnerabilities, in the worst case allowing execution of arbitrary system commands. Situations in which calls to system() have high risk include the following:

- When passing an unsanitized or improperly sanitized command string originating from a tainted source
- If a command is specified without a path name and the command processor path name resolution mechanism is accessible to an attacker
- If a relative path to an executable is specified and control over the current working directory is accessible to an attacker
- If the specified executable program can be spoofed by an attacker

Do not invoke a command processor via `system()` or equivalent functions to execute a command.

Noncompliant Code Example

In this noncompliant code example, the `system()` function is used to execute `any_cmd` in the host environment. Invocation of a command processor is not required.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

enum { BUFFERSIZE = 512 };

void func( const char *input) {
  char cmdbuf[BUFFERSIZE];
  int len_wanted = snprintf(cmdbuf, BUFFERSIZE,
                            "any_cmd '%s'", input);
  if (len_wanted >= BUFFERSIZE) {
    /* Handle error */
  } else if (len_wanted < 0) {
    /* Handle error */
  } else if ( system (cmdbuf) == -1) {
```

```
    /* Handle error */
  }
}
```

If this code is compiled and run with elevated privileges on a Linux system, for example, an attacker can create an account by entering the following string:

```
happy'; useradd 'attacker
```

The shell would interpret this string as two separate commands:

```
any_cmd 'happy';
useradd 'attacker'
```

and create a new user account that the attacker can use to access the compromised system.

This noncompliant code example also violates STR02-C. Sanitize data passed to complex subsystems.

Compliant Solution (POSIX)

In this compliant solution, the call to `system()` is replaced with a call to `execve()`. The `exec` family of functions does not use a full shell interpreter, so it is not vulnerable to command-injection attacks, such as the one illustrated in the noncompliant code example.

The `execlp()`, `execvp()`, and (nonstandard) `execvP()` functions duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a forward slash character (`/`). As a result, they should be used without a forward slash character (`/`) only if the `PATH` environment variable is set to a safe value, as described in ENV03-C. Sanitize the environment when invoking external programs.

The `execl()`, `execle()`, `execv()`, and `execve()` functions do not perform path name substitution.

Additionally, precautions should be taken to ensure the external executable cannot be modified by an untrusted user, for example, by ensuring the executable is not writable by the user.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

void func(char *input) {
  pid_t pid;
  int status;
  pid_t ret;
  char * const args[3] = {"any_exe", input, NULL};
  char **env;
  extern char **environ;

  /* ... Sanitize arguments ... */

  pid = fork();
  if (pid == -1) {
    /* Handle error */
  } else if (pid != 0) {
```

```
    while ((ret = waitpid(pid, &status, 0)) == -1) {
      if ( errno != EINTR) {
        /* Handle error */
        break ;
      }
    }
    if ((ret != -1) &&
      (!WIFEXITED(status) || !WEXITSTATUS(status)) ) {
      /* Report unexpected child status */
    }
  } else {
    /* ... Initialize env as a sanitized copy of environ ... */
    if (execve( "/usr/bin/any_cmd" , args, env) == -1) {
      /* Handle error */
      _Exit(127);
    }
  }
}
```

This compliant solution is significantly different from the preceding noncompliant code example. First, `input` is incorporated into the `args` array and passed as an argument to `execve()` , eliminating concerns about buffer overflow or string truncation while forming the command string. Second, this compliant solution forks a new process before executing `"/usr/bin/any_cmd"` in the child process. Although this method is more complicated than calling `system()` , the added security is worth the additional effort.

The exit status of 127 is the value set by the shell when a command is not found, and POSIX recommends that applications should do the same. XCU, Section 2.8.2, of *Standard for Information TechnologyPortable Operating System Interface (POSIX®), Base Specifications, Issue 7* [IEEE Std 1003.1:2013], says

> If a command is not found, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

Compliant Solution (Windows)

This compliant solution uses the Microsoft Windows `CreateProcess()` API:

```
#include <Windows.h>

void func( TCHAR *input) {
  STARTUPINFO si = { 0 };
  PROCESS_INFORMATION pi;
  si.cb = sizeof (si);
  if (!CreateProcess(TEXT( "any_cmd.exe" ), input, NULL, NULL, FALSE,
                     0, 0, 0, &si, &pi)) {
    /* Handle error */
  }
  CloseHandle(pi.hThread);
  CloseHandle(pi.hProcess);
}
```

This compliant solution relies on the `input` parameter being non- `const` . If it were `const` , the solution would need to create a copy of the parameter because the `CreateProcess()` function can modify the command-line arguments to be passed into the newly created process.

This solution creates the process such that the child process does not inherit any handles from the parent process, in compliance with WIN03-C. Understand HANDLE inheritance.

Noncompliant Code Example (POSIX)

This noncompliant code invokes the C `system()` function to remove the `.config` file in the user's home directory.

```
#include <stdlib.h>

void func( void ) {
  system ( "rm ~/.config" );
}
```

If the vulnerable program has elevated privileges, an attacker can manipulate the value of the `HOME` environment variable such that this program can remove any file named `.config` anywhere on the system.

Compliant Solution (POSIX)

An alternative to invoking the `system()` call to execute an external program to perform a required operation is to implement the functionality directly in the program using existing library calls. This compliant solution calls the POSIX `unlink()` function to remove a file without invoking the `system()` function [IEEE Std 1003.1:2013]

```
#include <pwd.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
void func( void ) {
  const char *file_format = "%s/.config" ;
  size_t len;
  char *pathname;
  struct passwd *pwd;

  /* Get /etc/passwd entry for current user */
  pwd = getpwuid(getuid());
  if (pwd == NULL) {
    /* Handle error */
  }

  /* Build full path name home dir from pw entry */

  len = strlen (pwd->pw_dir) + strlen (file_format) + 1;
  pathname = ( char *) malloc (len);
  if (NULL == pathname) {
    /* Handle error */
  }
  int r = snprintf(pathname, len, file_format, pwd->pw_dir);
  if (r < 0 || r >= len) {
    /* Handle error */
  }
  if (unlink(pathname) != 0) {
```

```
    /* Handle error */
  }

  free(pathname);
}
```

The `unlink()` function is not susceptible to a symlink attack where the final component of `pathname` (the file name) is a symbolic link because `unlink()` will remove the symbolic link and not affect any file or directory named by the contents of the symbolic link. (See FIO01-C. Be careful using functions that use file names for identification.) While this reduces the susceptibility of the `unlink()` function to symlink attacks, it does not eliminate it. The `unlink()` function is still susceptible if one of the directory names included in the `pathname` is a symbolic link. This could cause the `unlink()` function to delete a similarly named file in a different directory.

Compliant Solution (Windows)

This compliant solution uses the Microsoft Windows `SHGetKnownFolderPath()` API to get the current user's My Documents folder, which is then combined with the file name to create the path to the file to be deleted. The file is then removed using the `DeleteFile()` API.

```
#include <Windows.h>
#include <ShlObj.h>
#include <Shlwapi.h>

#if defined(_MSC_VER)
  #pragma comment(lib, "Shlwapi")
#endif

void func( void ) {
  HRESULT hr;
  LPWSTR path = 0;
  WCHAR full_path[MAX_PATH];

  hr = SHGetKnownFolderPath(&FOLDERID_Documents, 0, NULL, &path);
  if (FAILED(hr)) {
    /* Handle error */
  }
  if (!PathCombineW(full_path, path, L".config")) {
    /* Handle error */
  }
  CoTaskMemFree(path);
  if (!DeleteFileW(full_path)) {
    /* Handle error */
  }
}
```

Exceptions

ENV33-C-EX1: It is permissible to call `system()` with a null pointer argument to determine the presence of a command processor for the system.

Rule ERR33-C

Synopsis: Detect and handle standard library errors
Language: C++
Level: 1
Category: Security

Rule ERR54-CPP

Synopsis: Catch handlers should order their parameter types from most derived to least derived
Language: C++
Level: 1
Category: Security

Description

The C++ Standard, [except.handle], paragraph 4 [ISO/IEC 14882-2014], states the following:

> The handlers for a try block are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

Consequently, if two handlers catch exceptions that are derived from the same base class (such as `std::exception`), the most derived exception must come first.

Noncompliant Code Example

In this noncompliant code example, the first handler catches all exceptions of class `B` , as well as exceptions of class `D` , since they are also of class `B` . Consequently, the second handler does not catch any exceptions.

```
// Classes used for exception handling
class B {};
class D : public B {};

void f() {
  try {
    // ...
  } catch (B &b) {
    // ...
  } catch (D &d) {
    // ...
  }
}
```

Compliant Solution

In this compliant solution, the first handler catches all exceptions of class `D` , and the second handler catches all the other exceptions of class `B` .

```
// Classes used for exception handling
class B {};
```

```
class D : public B {};

void f() {
  try {
    // ...
  } catch (D &d) {
    // ...
  } catch (B &b) {
    // ...
  }
}
```

Rule EXP34-C

Synopsis: Do not dereference null pointers
Language: C++
Level: 1
Category: Security

Description

Dereferencing a null pointer is undefined behavior.

On many platforms, dereferencing a null pointer results in abnormal program termination, but this is not required by the standard. See "Clever Attack Exploits Fully-Patched Linux Kernel" [Goodin 2009] for an example of a code execution exploit that resulted from a null pointer dereference.

Noncompliant Code Example

This noncompliant code example is derived from a real-world example taken from a vulnerable version of the `libpng` library as deployed on a popular ARM-based cell phone [Jack 2007]. The `libpng` library allows applications to read, create, and manipulate PNG (Portable Network Graphics) raster image files. The `libpng` library implements its own wrapper to `malloc()` that returns a null pointer on error or on being passed a 0-byte-length argument.

This code also violates ERR33-C. Detect and handle standard library errors.

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, int length, const void *user_data) {
  png_charp chunkdata;
  chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
  /* ... */
  memcpy(chunkdata, user_data, length);
  /* ... */
 }
```

If `length` has the value `[unsupported char code: 8722]1` , the addition yields 0, and `png_malloc()` subsequently returns a null pointer, which is assigned to `chunkdata` . The `chunkdata` pointer is later used as a destination argument in a call to `memcpy()` , resulting in user-defined data overwriting memory starting at address 0. In the case of the ARM and XScale architectures, the `0x0` address is mapped in memory and serves as the exception vector table;

consequently, dereferencing `0x0` did not cause an abnormal program termination.

Compliant Solution

This compliant solution ensures that the pointer returned by `png_malloc()` is not null. It also uses the unsigned type `size_t` to pass the `length` parameter, ensuring that negative values are not passed to `func()`.

This solution also ensures that the `user_data` pointer is not null. Passing a null pointer to memcpy() would produce undefined behavior, even if the number of bytes to copy were 0. The `user_data` pointer could be invalid in other ways, such as pointing to freed memory. However there is no portable way to verify that the pointer is valid, other than checking for null.

```
#include <png.h> /* From libpng */
#include <string.h>

 void func(png_structp png_ptr, size_t length, const void *user_data) {
  png_charp chunkdata;
  if (length == SIZE_MAX) {
    /* Handle error */
  }
  chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
  if (NULL == chunkdata) {
    /* Handle error */
  }
  if (NULL == user_data) {
    /* Handle error */
  }
  /* ... */
  memcpy(chunkdata, user_data, length);
  /* ... */

 }
```

Noncompliant Code Example

In this noncompliant code example, `input_str` is copied into dynamically allocated memory referenced by `c_str`. If `malloc()` fails, it returns a null pointer that is assigned to `c_str`. When `c_str` is dereferenced in `memcpy()`, the program exhibits undefined behavior. Additionally, if `input_str` is a null pointer, the call to `strlen()` dereferences a null pointer, also resulting in undefined behavior. This code also violates ERR33-C. Detect and handle standard library errors.

```
#include <string.h>
#include <stdlib.h>

void f( const char *input_str) {
  size_t size = strlen(input_str) + 1;
  char *c_str = (char *) malloc(size);
  memcpy(c_str, input_str, size);
  /* ... */
  free(c_str);
  c_str = NULL;
  /* ... */
}
```

Compliant Solution

This compliant solution ensures that both `input_str` and the pointer returned by `malloc()` are not null:

```
#include <string.h>
#include <stdlib.h>

void f( const char *input_str) {
  size_t size;
  char *c_str;

  if (NULL == input_str) {
    /* Handle error */
  }

  size = strlen (input_str) + 1;
  c_str = ( char *) malloc (size);
  if (NULL == c_str) {
    /* Handle error */
  }
  memcpy (c_str, input_str, size);
  /* ... */
  free (c_str);
  c_str = NULL;
  /* ... */
}
```

Noncompliant Code Example

This noncompliant code example is from a version of `drivers/net/tun.c` and affects Linux kernel 2.6.30 [Goodin 2009]:

```
static unsigned int tun_chr_poll( struct file *file, poll_table *wait)  {
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk = tun->sk;
  unsigned int mask = 0;

  if (!tun)
    return POLLERR;

  DBG(KERN_INFO "%s: tun_chr_poll\n", tun->dev->name);

  poll_wait(file, &tun->socket.wait, wait);

  if (!skb_queue_empty(&tun->readq))
    mask |= POLLIN | POLLRDNORM;

  if (sock_writeable(sk) ||
     (!test_and_set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags) &&
     sock_writeable(sk)))
    mask |= POLLOUT | POLLWRNORM;
```

```
  if (tun->dev->reg_state != NETREG_REGISTERED)
    mask = POLLERR;

  tun_put(tun);
  return mask;
}
```

The `sk` pointer is initialized to `tun->sk` before checking if `tun` is a null pointer. Because null pointer dereferencing is undefined behavior, the compiler (GCC in this case) can optimize away the `if (!tun)` check because it is performed after `tun->sk` is accessed, implying that `tun` is non-null. As a result, this noncompliant code example is vulnerable to a null pointer dereference exploit, because null pointer dereferencing can be permitted on several platforms, for example, by using `mmap(2)` with the `MAP_FIXED` flag on Linux and Mac OS X, or by using the `shmat()` POSIX function with the `SHM_RND` flag [Liu 2009].

Compliant Solution

This compliant solution eliminates the null pointer deference by initializing `sk` to `tun->sk` following the null pointer check. It also adds assertions to document that certain other pointers must not be null.

```
static unsigned int tun_chr_poll( struct file *file, poll_table *wait)  {
  assert (file);
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk;
  unsigned int mask = 0;

  if (!tun)
    return POLLERR;
  assert (tun->dev);
  sk = tun->sk;
  assert (sk);
  assert (sk->socket);
  /* The remaining code is omitted because it is unchanged... */
}
```

Rule EXP53-CPP

Synopsis: Do not read uninitialized memory
Language: C++
Level: 1
Category: Security

Description

Local, automatic variables assume unexpected values if they are read before they are initialized. The C++ Standard, [dcl.init], paragraph 12 [ISO/IEC 14882-2014], states the following:

> If no initializer is specified for an object, the object is default-initialized. When storage for an object with automatic or dynamic storage duration is obtained, the object has an *indeterminate value*, and if no initialization is performed for the object, that object retains an indeterminate value until that value is replaced. If an indeterminate value is produced by an evaluation, the behavior is undefined except in the following cases:

> If an indeterminate value of unsigned narrow character type is produced by the evaluation of:
> the second or third operand of a conditional expression,
> the right operand of a comma expression,
> the operand of a cast or conversion to an unsigned narrow character type, or
> a discarded-value expression,
> then the result of the operation is an indeterminate value.
> If an indeterminate value of unsigned narrow character type is produced by the evaluation of the right operand of a simple assignment operator whose first operand is an lvalue of unsigned narrow character type, an indeterminate value replaces the value of the object referred to by the left operand.
> If an indeterminate value of unsigned narrow character type is produced by the evaluation of the initialization expression when initializing an object of unsigned narrow character type, that object is initialized to an indeterminate value.

The default initialization of an object is described by paragraph 7 of the same subclause:

> To *default-initialize* an object of type `T` means:
> if `T` is a (possibly cv-qualified) class type, the default constructor for `T` is called (and the initialization is ill-formed if `T` has no default constructor or overload resolution results in an ambiguity or in a function that is deleted or inaccessible from the context of the initialization);
> if `T` is an array type, each element is default-initialized;
> otherwise, no initialization is performed.
> If a program calls for the default initialization of an object of a const-qualified type `T`, `T` shall be a class type with a user-provided default constructor.

As a result, objects of type `T` with automatic or dynamic storage duration must be explicitly initialized before having their value read as part of an expression unless `T` is a class type or an array thereof or is an unsigned narrow character type. If `T` is an unsigned narrow character type, it may be used to initialize an object of unsigned narrow character type, which results in both objects having an indeterminate value. This technique can be used to implement copy operations such as `std::memcpy()` without triggering undefined behavior.

Additionally, memory dynamically allocated with a `new` expression is default-initialized when the *new-initialized* is omitted. Memory allocated by the standard library function `std::calloc()` is zero-initialized. Memory allocated by the standard library function `std::realloc()` assumes the values of the original pointer but may not initialize the full range of memory. Memory allocated by any other means (`std::malloc()`, allocator objects, `operator new()`, and so on) is assumed to be default-initialized.

Objects of static or thread storage duration are zero-initialized before any other initialization takes place [ISO/IEC 14882-2014] and need not be explicitly initialized before having their value read.

Reading uninitialized variables for creating entropy is problematic because these memory accesses can be removed by compiler optimization. VU925211 is an example of a vulnerability caused by this coding error [VU#925211].

Noncompliant Code Example

In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior.

```
#include <iostream>
```

```
void f() {
  int i;
  std::cout << i;
}
```

Compliant Solution

In this compliant solution, the object is initialized prior to printing its value.

```
#include <iostream>

void f() {
  int i = 0;
  std::cout << i;
}
```

Noncompliant Code Example

In this noncompliant code example, an `int *` object is allocated by a *new-expression*, but the memory it points to is not initialized. The object's pointer value and the value it points to are printed to the standard output stream. Printing the pointer value is well-defined, but attempting to print the value pointed to yields an indeterminate value, resulting in undefined behavior.

```
#include <iostream>

void f() {
  int *i = new int;
  std::cout << i << ", " << *i;
}
```

Compliant Solution

In this compliant solution, the memory is direct-initialized to the value `12` prior to printing its value.

```
#include <iostream>

void f() {
  int *i = new int(12);
  std::cout << i << ", " << *i;
}
```

Initialization of an object produced by a *new-expression* is performed by placing (possibly empty) parenthesis or curly braces after the type being allocated. This causes direct initialization of the pointed-to object to occur, which will zero-initialize the object if the initialization omits a value, as illustrated by the following code.

```
int *i = new int(); // zero-initializes *i
int *j = new int{}; // zero-initializes *j
int *k = new int( 12 ); // initializes *k to 12
int *l = new int{ 12 }; // initializes *l to 12
```

Noncompliant Code Example

In this noncompliant code example, the class member variable `c` is not explicitly initialized by a *ctor-initializer* in the default constructor. Despite the local variable `s` being default-initialized, the use of

`c` within the call to `S::f()` results in the evaluation of an object with indeterminate value, resulting in undefined behavior.

```
class S {
   int c;

public :
   int f( int i) const { return i + c; }
};

void f() {
   S s;
   int i = s.f(10);
}
```

Compliant Solution

In this compliant solution, `S` is given a default constructor that initializes the class member variable `c.`

```
class S {
   int c;

public :
   S() : c(0) {}
   int f( int i) const { return i + c; }
};

void f() {
   S s;
   int i = s.f(10);
}
```

Rule EXT01-CPP

Synopsis: The called function is unsafe for security related code
Language: C++
Level: 3
Category: Security

Description

Examples of unsafe functions are:

- I/O functions that could cause a buffer overflow such as `scanf`, `fscanf` and `gets`.
- String buffer access functions that could cause a buffer overflow such as `sprintf`, `sscanf`, `strcat`, `strcpy`, and `__builtin___sprintf_chk`.
- Pseudo-random number generation functions for which it is too easy to break the encryption such as `initstate`, `lcong48`, `rand`, `random`, `seed48`, `setstate`, and `[dejlmn]rand48`.

Rule EXT02-CPP

Synopsis: non-constant printf format string may be susceptible to format string attacks
Language: C++
Level: 3
Category: Security

Rule EXT03-CPP

Synopsis: Calling a function which may pose a security risk if it is used inappropriately
Language: C++
Level: 3
Category: Security

Rule EXT04-CPP

Synopsis: Using an insecure temporary file creation function
Language: C++
Level: 3
Category: Security

Description

This rule is about cases where a temporary file is created in an insecure manner. When that happens in a program that runs with elevated privileges, the program is vulnerable to race condition attacks and can be used to subvert system security.

Many programs create temporary files in shared directories such as `/tmp`. There are C library routines that assist in creating unique temporary files, but many of them are insecure as they make a program vulnerable to race condition attacks.

If the name of a temporary file is easily guessed, or the filename is used unsafely after temp file creation, or the umask is not safely set before calling a safe routine, an attacker can take control of a vulnerable application and system.

Avoid using insecure temporary file creation routines. Instead, use `mkstemp()` for creating temp files. When using `mkstemp()`, remember to safely set the umask before to restrict the resulting temporary file permissions to only the owner. Also, do not pass on the filename to another privileged system call. Use the returned file descriptor instead.

The following example generates a defect because `mktemp()` is insecure: it is easy to guess the name of the temporary file it creates. Similar functions include `tmpnam()`, `tempnam()`, and `tmpfile()`.

```
void secure_temp_example() {
    char *tmp, *tmp2, *tmp3;
    char buffer[1024];
    tmp = mktemp(buffer);
}
```

Rule EXT05-CPP

Synopsis: A user-land pointer is dereferenced without safety checks in the kernel
Language: C++
Level: 3
Category: Security

Description

Be aware of cases where an operating system kernel unsafely dereferences user pointers. Operating systems cannot directly dereference user-space pointers safely. Instead, they must access the pointed-to data using special "paranoid" routines (for example: using the `copyin()` and `copyout()` functions on BSD derived systems, or the `copy_from_user()` and `copy_to_user()` functions on Linux derived systems). A single unsafe dereference can crash the system, allow unauthorized reading/writing of kernel memory, or give a malicious party complete system control.

The following example has a defect because `pstr` is correctly copied in from user space with the `copyin()` method, but its field `ps_argvstr`, another pointer to user space memory, is unsafely dereferenced by the expression `pstr.ps_argvstr[i]`.

```
void user_pointer_example() {
    error = copyin((void *)p->p_sysent->sv_psstrings, &pstr, sizeof(pstr));
    if (error)
        return (error);
    for (i = 0; i < pstr.ps_nargvstr; i++) {
        sbuf_copyin(sb, pstr.ps_argvstr[i], 0);
        sbuf_printf(sb, "%c", '\0');
    }
}
```

Rule FIO30-C

Synopsis: Exclude user input from format strings
Language: C++
Level: 1
Category: Security

Description

Never call a formatted I/O function with a format string containing a tainted value . An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process [Seacord 2013b]. Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. For example, formatted output functions can be used to write an integer value to a specified address using the %n conversion specifier.

Noncompliant Code Example

The `incorrect_password()` function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by `user` . This is

an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to `stderr` using the C Standard `fprintf()` function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password( const char *user) {
  int ret;
  /* User names are restricted to 256 or fewer characters */
  static const char msg_format[] = "%s cannot be authenticated.\n" ;
  size_t len = strlen (user) + sizeof (msg_format);
  char *msg = ( char *) malloc (len);
  if (msg == NULL) {
    /* Handle error */
  }
  ret = snprintf(msg, len, msg_format, user);
  if (ret < 0) {
    /* Handle error */
  } else if (ret >= len) {
    /* Handle truncated output */
  }
  fprintf (stderr, msg);
  free (msg);
}
```

The `incorrect_password()` function calculates the size of the message, allocates dynamic storage, and then constructs the message in the allocated memory using the `snprintf()` function. The addition operations are not checked for integer overflow because the string referenced by `user` is known to have a length of 256 or less. Because the `%s` characters are replaced by the string referenced by `user` in the call to `snprintf()` , the resulting string needs 1 byte less than is allocated. The `snprintf()` function is commonly used for messages that are displayed in multiple locations or messages that are difficult to build. However, the resulting code contains a format-string vulnerability because the `msg` includes untrusted user input and is passed as the format-string argument in the call to `fprintf()` .

Compliant Solution (`fputs()`)

This compliant solution fixes the problem by replacing the `fprintf()` call with a call to `fputs()` , which outputs `msg` directly to `stderr` without evaluating its contents:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password( const char *user) {
  int ret;
  /* User names are restricted to 256 or fewer characters */
  static const char msg_format[] = "%s cannot be authenticated.\n" ;
  size_t len = strlen (user) + sizeof (msg_format);
  char *msg = ( char *) malloc (len);
  if (msg == NULL) {
    /* Handle error */
  }
  ret = snprintf(msg, len, msg_format, user);
  if (ret < 0) {
```

```
    /* Handle error */
  } else if (ret >= len) {
    /* Handle truncated output */
  }
  fputs(msg, stderr);
  free(msg);
}
```

Compliant Solution (`fprintf()`)

This compliant solution passes the untrusted user input as one of the variadic arguments to `fprintf()` and not as part of the format string, eliminating the possibility of a format-string vulnerability:

```
#include <stdio.h>

void incorrect_password(const char *user) {
  static const char msg_format[] = "%s cannot be authenticated.\n";
  fprintf(stderr, msg_format, user);
}
```

Noncompliant Code Example (POSIX)

This noncompliant code example is similar to the first noncompliant code example but uses the POSIX function `syslog()` [IEEE Std 1003.1:2013] instead of the `fprintf()` function. The `syslog()` function is also susceptible to format-string vulnerabilities.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>

void incorrect_password(const char *user) {
  int ret;
  /* User names are restricted to 256 or fewer characters */
  static const char msg_format[] = "%s cannot be authenticated.\n";
  size_t len = strlen(user) + sizeof(msg_format);
  char *msg = (char *)malloc(len);
  if (msg == NULL) {
    /* Handle error */
  }
  ret = snprintf(msg, len, msg_format, user);
  if (ret < 0) {
    /* Handle error */
  } else if (ret >= len) {
    /* Handle truncated output */
  }
  syslog(LOG_INFO, msg);
  free(msg);
}
```

The `syslog()` function first appeared in BSD 4.2 and is supported by Linux and other modern UNIX implementations. It is not available on Windows systems.

Compliant Solution (POSIX)

This compliant solution passes the untrusted user input as one of the variadic arguments to `syslog()` instead of including it in the format string:

```
#include <syslog.h>

void incorrect_password( const char *user) {
  static const char msg_format[] = "%s cannot be authenticated.\n";
  syslog(LOG_INFO, msg_format, user);
}
```

Rule FIO34-C

Synopsis: Distinguish between characters read from a file and EOF or WEOF
Language: C++
Level: 1
Category: Security

Description

The `EOF` macro represents a negative value that is used to indicate that the file is exhausted and no data remains when reading data from a file. `EOF` is an example of an in-band error indicator. In-band error indicators are problematic to work with, and the creation of new in-band-error indicators is discouraged by ERR02-C. Avoid in-band error indicators.

The byte I/O functions `fgetc()` , `getc()` , and `getchar()` all read a character from a stream and return it as an `int.` (See STR00-C. Represent characters using an appropriate type.) If the stream is at the end of the file, the end-of-file indicator for the stream is set and the function returns `EOF` . If a read error occurs, the error indicator for the stream is set and the function returns `EOF` . If these functions succeed, they cast the character returned into an `unsigned char` .

Because `EOF` is negative, it should not match any unsigned character value. However, this is only true for implementations where the `int` type is wider than `char` . On an implementation where `int` and `char` have the same width, a character-reading function can read and return a valid character that has the same bit-pattern as `EOF` . This could occur, for example, if an attacker inserted a value that looked like `EOF` into the file or data stream to alter the behavior of the program.

The C Standard requires only that the `int` type be able to represent a maximum value of +32767 and that a `char` type be no larger than an `int` . Although uncommon, this situation can result in the integer constant expression `EOF` being indistinguishable from a valid character; that is, `(int)(unsigned char)65535 == -1` . Consequently, failing to use `feof()` and `ferror()` to detect end-of-file and file errors can result in incorrectly identifying the `EOF` character on rare implementations where `sizeof(int) == sizeof(char)` .

This problem is much more common when reading wide characters. The `fgetwc()` , `getwc()` , and `getwchar()` functions return a value of type `wint_t` . This value can represent the next wide character read, or it can represent `WEOF` , which indicates end-of-file for wide character streams. On most implementations, the `wchar_t` type has the same width as `wint_t` , and these functions can return a character indistinguishable from `WEOF` .

In the UTF-16 character set, `0xFFFF` is guaranteed not to be a character, which allows `WEOF` to be represented as the value `-1` . Similarly, all UTF-32 characters are positive when viewed as a signed 32-bit integer. All widely used character sets are designed with at least one value that does not represent a character. Consequently, it would require a custom character set designed without consideration of the C programming language for this problem to occur with wide characters or with ordinary characters that are as wide as `int` .

The C Standard `feof()` and `ferror()` functions are not subject to the problems associated with character and integer sizes and should be used to verify end-of-file and file errors for susceptible implementations [Kettlewell 2002]. Calling both functions on each iteration of a loop adds significant overhead, so a good strategy is to temporarily trust `EOF` and `WEOF` within the loop but verify them with `feof()` and `ferror()` following the loop.

Noncompliant Code Example

This noncompliant code example loops while the character `c` is not `EOF` :

```
#include <stdio.h>

void func( void ) {
  int c;

  do {
    c = getchar ();
  } while (c != EOF);
}
```

Although `EOF` is guaranteed to be negative and distinct from the value of any unsigned character, it is not guaranteed to be different from any such value when converted to an `int` . Consequently, when `int` has the same width as `char` , this loop may terminate prematurely.

Compliant Solution (Portable)

This compliant solution uses `feof()` to test for end-of-file and `ferror()` to test for errors:

```
#include <stdio.h>

void func( void ) {
  int c;

  do {
    c = getchar ();
  } while (c != EOF);
  if ( feof (stdin)) {
    /* Handle end of file */
  } else if ( ferror (stdin)) {
    /* Handle file error */
  } else {
    /* Received a character that resembles EOF; handle error */
  }
}
```

Noncompliant Code Example (Nonportable)

This noncompliant code example uses an assertion to ensure that the code is executed only on architectures where `int` is wider than `char` and `EOF` is guaranteed not to be a valid character value. However, this code example is noncompliant because the variable `c` is declared as a `char` rather than an `int` , making it possible for a valid character value to compare equal to the value of the `EOF` macro when `char` is signed because of sign extension:

```
#include <assert.h>
#include <limits.h>
#include <stdio.h>

void func( void ) {
  char c;
  static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation" );

  do {
    c = getchar ();
  } while (c != EOF);
}
```

Assuming that a `char` is a signed 8-bit type and an int is a 32-bit type, if `getchar()` returns the character value `'\xff` (decimal 255), it will be interpreted as `EOF` because this value is sign-extended to `0xFFFFFFFF` (the value of `EOF`) to perform the comparison. (See STR34-C. Cast characters to unsigned char before converting to larger integer sizes.)

Compliant Solution (Nonportable)

This compliant solution declares `c` to be an `int` . Consequently, the loop will terminate only when the file is exhausted.

```
#include <assert.h>
#include <stdio.h>
#include <limits.h>

void func( void ) {
  int c;
  static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation" );

  do {
    c = getchar ();
  } while (c != EOF);
}
```

Noncompliant Code Example (Wide Characters)

In this noncompliant example, the result of the call to the C standard library function `getwc()` is stored into a variable of type `wchar_t` and is subsequently compared with `WEOF` :

```
#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum { BUFFER_SIZE = 32 };
```

```
void g( void ) {
  wchar_t buf[BUFFER_SIZE];
  wchar_t wc;
  size_t i = 0;

  while ((wc = getwc(stdin)) != L '\n' && wc != WEOF) {
    if (i < (BUFFER_SIZE - 1)) {
      buf[i++] = wc;
    }
  }
  buf[i] = L '\0' ;
}
```

This code suffers from two problems. First, the value returned by `getwc()` is immediately converted to `wchar_t` before being compared with `WEOF` . Second, there is no check to ensure that `wint_t` is wider than `wchar_t` . Both of these problems make it possible for an attacker to terminate the loop prematurely by supplying the wide-character value matching `WEOF` in the file.

Compliant Solution (Portable)

This compliant solution declares `c` to be a `wint_t` to match the integer type returned by `getwc()` . Furthermore, it does not rely on `WEOF` to determine end-of-file definitively.

```
#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum {BUFFER_SIZE = 32 }

void g( void ) {
  wchar_t buf[BUFFER_SIZE];
  wint_t wc;
  size_t i = 0;

  while ((wc = getwc(stdin)) != L '\n' && wc != WEOF) {
    if (i < BUFFER_SIZE - 1) {
      buf[i++] = wc;
    }
  }

  if ( feof (stdin) || ferror (stdin)) {
   buf[i] = L '\0' ;
  } else {
    /* Received a wide character that resembles WEOF; handle error */
  }
}
```

Exceptions

FIO34-C-EX1: A number of C functions do not return characters but can return `EOF` as a status code. These functions include `fclose()` , `fflush()` , `fputs()` , `fscanf()` , `puts()` , `scanf()` , `sscanf()` , `vfscanf()` , and `vscanf()` . These return values can be compared to `EOF` without validating the result.

Rule FIO37-C

Synopsis: Do not assume that fgets() or fgetws() returns a nonempty string when successful
Language: C++
Level: 1
Category: Security

Description

Errors can occur when incorrect assumptions are made about the type of data being read. These assumptions may be violated, for example, when binary data has been read from a file instead of text from a user's terminal or the output of a process is piped to `stdin`. (See FIO14-C. Understand the difference between text mode and binary mode with file streams.) On some systems, it may also be possible to input a null byte (as well as other binary codes) from the keyboard.

Subclause 7.21.7.2 of the C Standard [ISO/IEC 9899:2011] says,

> The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned.

The wide-character function `fgetws()` has the same behavior. Therefore, if `fgets()` or `fgetws()` returns a non-null pointer, it is safe to assume that the array contains data. However, it is erroneous to assume that the array contains a nonempty string because the data may contain null characters.

Noncompliant Code Example

This noncompliant code example attempts to remove the trailing newline (`\n`) from an input line. The `fgets()` function is typically used to read a newline-terminated line of input from a stream. It takes a size parameter for the destination buffer and copies, at most, `size - 1` characters from a stream to a character array.

```
#include <stdio.h>
#include <string.h>

enum { BUFFER_SIZE = 1024 };

void func( void ) {
  char buf[BUFFER_SIZE];

  if ( fgets (buf, sizeof (buf), stdin) == NULL) {
    /* Handle error */
  }
  buf[ strlen (buf) - 1] = '\0' ;
}
```

The `strlen()` function computes the length of a string by determining the number of characters that precede the terminating null character. A problem occurs if the first character read from the input by `fgets()` happens to be a null character. This may occur, for example, if a binary data file is read by the `fgets()` call [Lai 2006]. If the first character in `buf` is a null character, `strlen(buf)` returns 0, the expression `strlen(buf) - 1` wraps around to a large positive value, and a write-outside-array-bounds error occurs.

Compliant Solution

This compliant solution uses `strchr()` to replace the newline character in the string if it exists:

```
#include <stdio.h>
#include <string.h>

enum { BUFFER_SIZE = 1024 };

void func( void ) {
  char buf[BUFFER_SIZE];
  char *p;

  if ( fgets (buf, sizeof (buf), stdin)) {
    p = strchr (buf, '\n' );
    if (p) {
      *p = '\0' ;
    }
  } else {
    /* Handle error */
  }
}
```

Rule FIO45-C

Synopsis: Avoid TOCTOU race conditions while accessing files
Language: C++
Level: 2
Category: Security

Description

A TOCTOU (time-of-check, time-of-use) race condition is possible when two or more concurrent processes are operating on a shared file system [Seacord 2013b]. Typically, the first access is a check to verify some attribute of the file, followed by a call to use the file. An attacker can alter the file between the two accesses, or replace the file with a symbolic or hard link to a different file. These TOCTOU conditions can be exploited when a program performs two or more file operations on the same file name or path name.

A program that performs two or more file operations on a single file name or path name creates a race window between the two file operations. This race window comes from the assumption that the file name or path name refers to the same resource both times. If an attacker can modify the file, remove it, or replace it with a different file, then this assumption will not hold.

Noncompliant Code Example

If an existing file is opened for writing with the `w` mode argument, the file's previous contents (if any) are destroyed. This noncompliant code example tries to prevent an existing file from being overwritten by first opening it for reading before opening it for writing. An attacker can exploit the race window between the two calls to `fopen()` to overwrite an existing file.

```
#include <stdio.h>

void open_some_file( const char *file) {
  FILE *f = fopen (file, "r" );
  if (NULL != f) {
    /* File exists, handle error */
  } else {
    if ( fclose (f) == EOF) {
      /* Handle error */
    }
    f = fopen (file, "w" );
    if (NULL == f) {
      /* Handle error */
    }

    /* Write to file */
    if ( fclose (f) == EOF) {
      /* Handle error */
    }
  }
}
```

Compliant Solution

This compliant solution invokes `fopen()` at a single location and uses the `x` mode of `fopen()`, which was added in C11. This mode causes `fopen()` to fail if the file exists. This check and subsequent open is performed without creating a race window. The `x` mode provides exclusive access to the file only if the host environment provides this support.

```
#include <stdio.h>

void open_some_file( const char *file) {
  FILE *f = fopen (file, "wx" );
  if (NULL == f) {
    /* Handle error */
  }
  /* Write to file */
  if ( fclose (f) == EOF) {
    /* Handle error */
  }
}
```

Compliant Solution (POSIX)

This compliant solution uses the `O_CREAT` and `O_EXCL` flags of POSIX's `open()` function. These flags cause `open()` to fail if the file exists.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void open_some_file( const char *file) {
  int fd = open(file, O_CREAT | O_EXCL | O_WRONLY);
  if (-1 != fd) {
    FILE *f = fdopen(fd, "w" );
```

```
    if (NULL != f) {
      /* Write to file */

      if ( fclose (f) == EOF) {
        /* Handle error */
      }
    } else {
      if (close(fd) == -1) {
        /* Handle error */
      }
    }
  }
}
```

Exceptions

FIO45-C-EX1: TOCTOU race conditions require that the vulnerable process is more privileged than the attacker; otherwise there is nothing to be gained from a successful attack. An unprivileged process is not subject to this rule.

FIO45-C-EX2: Accessing a file name or path name multiple times is permitted if the file referenced resides in a secure directory. (For more information, see FIO15-C. Ensure that file operations are performed in a secure directory.)

FIO45-C-EX3: Accessing a file name or path name multiple times is permitted if the program can verify that every operation operates on the same file.

This POSIX code example verifies that each subsequent file access operates on the same file. In POSIX, every file can be uniquely identified by using its device and i-node attributes. This code example checks that a file name refers to a regular file (and not a directory, symbolic link, or other special file) by invoking `lstat()` . This call also retrieves its device and i-node. The file is subsequently opened. Finally, the program verifies that the file that was opened is the same one (matching device and i-nodes) as the file that was confirmed as a regular file.

```
#include <sys/stat.h>
#include <fcntl.h>

int open_regular_file( char *filename, int flags) {
  struct stat lstat_info;
  struct stat fstat_info;
  int f;

  if (lstat(filename, &lstat_info) == -1) {
    /* File does not exist, handle error */
  }

  if (!S_ISREG(lstat_info.st_mode)) {
    /* File is not a regular file, handle error */
  }

  if ((f = open(filename, flags)) == -1) {
    /* File has disappeared, handle error */
  }
```

```
  if (fstat(f, &fstat_info) == -1) {
    /* Handle error */
  }

  if (lstat_info.st_ino != fstat_info.st_ino  ||
      lstat_info.st_dev != fstat_info.st_dev) {
    /* Open file is not the expected regular file, handle error */
  }

  /* f is the expected regular open file */
  return f;
}
```

Rule MEM50-CPP

Synopsis: Do not access freed memory
Language: C++
Level: 1
Category: Security

Description

Evaluating a pointer including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignmentinto memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable vulnerabilities.

It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed.

Noncompliant Code Example (`new` and `delete`)

In this noncompliant code example, `s` is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems.

```
#include <new>

struct S {
  void f();
};

void g() noexcept( false ) {
  S *s = new S;
  // ...
  delete s;
  // ...
```

```
  s->f();
}
```

The function `g()` is marked `noexcept(false)` to comply with MEM52-CPP. Detect and handle memory allocation errors.

Compliant Solution (`new` and `delete`)

In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required.

```
#include <new>

struct S {
  void f();
};

void g() noexcept( false ) {
  S *s = new S;
  // ...
  s->f();
  delete s;
}
```

Compliant Solution (Automatic Storage Duration)

When possible, use automatic storage duration instead of dynamic storage duration. Since `s` is not required to live beyond the scope of `g()` , this compliant solution uses automatic storage duration to limit the lifetime of `s` to the scope of `g().`

```
struct S {
  void f();
};

void g() {
  S s;
  // ...
  s.f();
}
```

Noncompliant Code Example (`std::unique_ptr`)

In the following noncompliant code example, the dynamically allocated memory managed by the `buff` object is accessed after it has been implicitly deallocated by the object's destructor.

```
#include <iostream>
#include <memory>
#include <cstring>

int main( int argc, const char *argv[]) {
  const char *s = "" ;
  if (argc > 1) {
    enum { BufferSize = 32 };
    try {
      std::unique_ptr< char []> buff( new char [BufferSize]);
```

```
      std::memset(buff.get(), 0, BufferSize);
      // ...
      s = std::strncpy(buff.get(), argv[1], BufferSize - 1);
    }catch(std::bad_alloc &) {
      // Handle error
    }
  }

  std::cout << s << std::endl;
}
```

This code always creates a null-terminated byte string, despite its use of `strncpy()`, because it leaves the final `char` in the buffer set to 0.

Compliant Solution (`std::unique_ptr`)

In this compliant solution, the lifetime of the `buff` object extends past the point at which the memory managed by the object is accessed.

```
#include <iostream>
#include <memory>
#include <cstring>

int main(int argc, const char *argv[]) {
  std::unique_ptr<char []> buff;
  const char *s = "";

  if(argc > 1) {
    enum { BufferSize = 32 };
    try {
      buff.reset(new char [BufferSize]);
      std::memset(buff.get(), 0, BufferSize);
      // ...
      s = std::strncpy(buff.get(), argv[1], BufferSize - 1);
    }catch(std::bad_alloc &) {
      // Handle error
    }
  }

  std::cout << s << std::endl;
}
```

Compliant Solution

In this compliant solution, a variable with automatic storage duration of type `std::string` is used in place of the `std::unique_ptr<char[]>`, which reduces the complexity and improves the security of the solution.

```
#include <iostream>
#include <string>

int main(int argc, const char *argv[]) {
  std::string str;

  if(argc > 1) {
```

```
    str = argv[1];
  }

  std::cout << str << std::endl;
}
```

Noncompliant Code Example (`std::string::c_str()`)

In this noncompliant code example, `std::string::c_str()` is being called on a temporary `std::string` object. The resulting pointer will point to released memory once the `std::string` object is destroyed at the end of the assignment expression, resulting in undefined behavior when accessing elements of that pointer.

```
#include <string>

std::string str_func();
void display_string( const char *);

void f() {
  const char *str = str_func().c_str();
  display_string(str);  /* Undefined behavior */
}
```

Compliant solution (`std::string::c_str()`)

In this compliant solution, a local copy of the string returned by `str_func()` is made to ensure that string `str` will be valid when the call to `display_string()` is made.

```
#include <string>

std::string str_func();
void display_string( const char *s);

void f() {
  std::string str = str_func();
  const char *cstr = str.c_str();
  display_string(cstr);  /* ok */
}
```

Noncompliant Code Example

In this noncompliant code example, an attempt is made to allocate zero bytes of memory through a call to `operator new()`. If this request succeeds, `operator new()` is required to return a non-null pointer value. However, according to the C++ Standard, [basic.stc.dynamic.allocation], paragraph 2 [ISO/IEC 14882-2014], attempting to dereference memory through such a pointer results in undefined behavior.

```
#include <new>

void f() noexcept( false ) {
  unsigned char *ptr = static_cast <unsigned char *>(::operator new (0));
  *ptr = 0;
  // ...
  ::operator delete (ptr);
}
```

Compliant Solution

The compliant solution depends on programmer intent. If the programmer intends to allocate a single `unsigned char` object, the compliant solution is to use `new` instead of a direct call to `operator new()`, as this compliant solution demonstrates.

```
void f() noexcept( false ) {
  unsigned char *ptr = new unsigned char;
  *ptr = 0;
  // ...
  delete ptr;
}
```

Compliant Solution

If the programmer intends to allocate zero bytes of memory (perhaps to obtain a unique pointer value that cannot be reused by any other pointer in the program until it is properly released), then instead of attempting to dereference the resulting pointer, the recommended solution is to declare `ptr` as a `void *`, which cannot be dereferenced by a conforming implementation.

```
#include <new>

void f() noexcept( false ) {
  void *ptr = ::operator new (0);
  // ...
  ::operator delete (ptr);
}
```

Rule MEM56-CPP

Synopsis: Do not store an already-owned pointer value in an unrelated smart pointer
Language: C++
Level: 1
Category: Security

Description

Smart pointers such as `std::unique_ptr` and `std::shared_ptr` encode pointer ownership semantics as part of the type system. They wrap a pointer value, provide pointer-like semantics through `operator *()` and `operator->()` member functions, and control the lifetime of the pointer they manage. When a smart pointer is constructed from a pointer value, that value is said to be *owned* by the smart pointer.

Calling `std::unique_ptr::release()` will relinquish ownership of the managed pointer value. Destruction of, move assignment of, or calling `std::unique_ptr::reset()` on a `std::unique_ptr` object will also relinquish ownership of the managed pointer value, but results in destruction of the managed pointer value. If a call to `std::shared_ptr::unique()` returns true, then destruction of or calling `std::shared_ptr::reset()` on that `std::shared_ptr` object will relinquish ownership of the managed pointer value but results in destruction of the managed pointer value.

Some smart pointers, such as `std::shared_ptr`, allow multiple smart pointer objects to manage the

same underlying pointer value. In such cases, the initial smart pointer object owns the pointer value, and subsequent smart pointer objects are related to the original smart pointer. Two smart pointers are *related* when the initial smart pointer is used in the initialization of the subsequent smart pointer objects. For instance, copying a `std::shared_ptr` object to another `std::shared_ptr` object via copy assignment creates a relationship between the two smart pointers, whereas creating a `std::shared_ptr` object from the managed pointer value of another `std::shared_ptr` object does not.

Do not create an unrelated smart pointer object with a pointer value that is owned by another smart pointer object. This includes resetting a smart pointer's managed pointer to an already-owned pointer value, such as by calling `reset()`.

Noncompliant Code Example

In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable `p2` is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable `p1` is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability.

```
#include <memory>

void f() {
  int *i = new int;
  std::shared_ptr<int> p1(i);
  std::shared_ptr<int> p2(i);
}
```

Compliant Solution

In this compliant solution, the `std::shared_ptr` objects are related to one another through copy construction. When the local, automatic variable `p2` is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable `p1` is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls `std::make_shared` () instead of allocating a raw pointer and storing its value in a local variable.

```
#include <memory>

void f() {
  std::shared_ptr<int> p1 = std::make_shared<int>();
  std::shared_ptr<int> p2(p1);
}
```

Noncompliant Code Example

In this noncompliant code example, the `poly` pointer value owned by a `std::shared_ptr` object is cast to the `D *` pointer type with `dynamic_cast` in an attempt to obtain a `std::shared_ptr` of the polymorphic derived type. However, this eventually results in undefined behavior as the same pointer is thereby stored in two different `std::shared_ptr` objects. When `g()` exits, the pointer stored in `derived` is freed by the default deleter. Any further use of `poly` results in accessing freed memory. When `f()` exits, the same pointer stored in `poly` is destroyed, resulting in a double-free vulnerability.

```
#include <memory>
```

```
struct B {
  virtual ~B() = default; // Polymorphic object
  // ...
};
struct D : B {};

void g(std::shared_ptr<D> derived);

void f() {
  std::shared_ptr<B> poly( new D);
  // ...
  g(std::shared_ptr<D>( dynamic_cast <D *>(poly.get())));
  // Any use of poly will now result in accessing freed memory.
}
```

Compliant Solution

In this compliant solution, the `dynamic_cast` is replaced with a call to `std::dynamic_pointer_cast()`, which returns a `std::shared_ptr` of the polymorphic type with the valid shared pointer value. When `g()` exits, the reference count to the underlying pointer is decremented by the destruction of `derived`, but because of the reference held by `poly` (within `f()`), the stored pointer value is still valid after `g()` returns.

```
#include <memory>

struct B {
  virtual ~B() = default; // Polymorphic object
  // ...
};
struct D : B {};

void g(std::shared_ptr<D> derived);

void f() {
  std::shared_ptr<B> poly( new D);
  // ...
  g(std::dynamic_pointer_cast<D, B>(poly));
  // poly is still referring to a valid pointer value.
}
```

Noncompliant Code Example

In this noncompliant code example, a `std::shared_ptr` of type `S` is constructed and stored in `s1`. Later, `S::g()` is called to get another shared pointer to the pointer value managed by `s1`. However, the smart pointer returned by `S::g()` is not related to the smart pointer stored in `s1`. When `s2` is destroyed, it will free the pointer managed by `s1`, causing a double-free vulnerability when `s1` is destroyed.

```
#include <memory>

struct S {
  std::shared_ptr<S> g() { return std::shared_ptr<S>( this ); }
};

void f() {
```

```
  std::shared_ptr<S> s1 = std::make_shared<S>();
  // ...
  std::shared_ptr<S> s2 = s1->g();
}
```

Compliant Solution

The compliant solution is to use `std::enable_shared_from_this::shared_from_this()` to get a shared pointer from `S` that is related to an existing `std::shared_ptr` object. A common implementation strategy is for the `std::shared_ptr` constructors to detect the presence of a pointer that inherits from `std::enable_shared_from_this`, and automatically update the internal bookkeeping required for `std::enable_shared_from_this::shared_from_this()` to work. Note that `std::enable_shared_from_this::shared_from_this()` requires an existing `std::shared_ptr` instance that manages the pointer value pointed to by `this`. Failure to meet this requirement results in undefined behavior, as it would result in a smart pointer attempting to manage the lifetime of an object that itself does not have lifetime management semantics.

```
#include <memory>

struct S : std::enable_shared_from_this<S> {
  std::shared_ptr<S> g() { return shared_from_this(); }
};

void f() {
  std::shared_ptr<S> s1 = std::make_shared<S>();
  std::shared_ptr<S> s2 = s1->g();
}
```

Rule MSC30-C

Synopsis: Do not use the rand() function for generating pseudorandom numbers
Language: C++
Level: 2
Category: Security

Description

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.

The C Standard `rand()` function makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of `rand()` have a comparatively short cycle and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs.

Noncompliant Code Example

The following noncompliant code generates an ID with a numeric part produced by calling the `rand()` function. The IDs produced are predictable and have limited randomness.

```
#include <stdio.h>
#include <stdlib.h>
```

```
enum { len = 12 };

void func( void ) {
   /*
    * id will hold the ID, starting with the characters
    *  "ID" followed by a random integer.
    */
   char id[len];
   int r;
   int num;
   /* ... */
   r = rand (); /* Generate a random integer */
   num = snprintf(id, len, "ID%-d", r); /* Generate the ID */
   /* ... */
}
```

Compliant Solution (POSIX)

This compliant solution replaces the `rand()` function with the POSIX `random()` function:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum { len = 12 };

void func( void ) {
   /*
    * id will hold the ID, starting with the characters
    *  "ID" followed by a random integer.
    */
   char id[len];
   int r;
   int num;
   /* ... */
   struct timespec ts;
   if (timespec_get(&ts, TIME_UTC) == 0) {
     /* Handle error */
   }
   srandom(ts.tv_nsec ^ ts.tv_sec); /* Seed the PRNG */
   /* ... */
   r = random(); /* Generate a random integer */
   num = snprintf(id, len, "ID%-d", r); /* Generate the ID */
   /* ... */
}
```

The POSIX `random()` function is a better pseudorandom number generator. Although on some platforms the low dozen bits generated by `rand()` go through a cyclic pattern, all the bits generated by `random()` are usable. The `rand48` family of functions provides another alternative for pseudorandom numbers.

Although not specified by POSIX, `arc4random()` is another possibility for systems that support it. The `arc4random(3)` manual page [OpenBSD] states

> ... provides higher quality of data than those described in rand(3), random(3), and drand48(3).

To achieve the best random numbers possible, an implementation-specific function must be used. When unpredictability is crucial and speed is not an issue, as in the creation of strong cryptographic keys, use a true entropy source, such as `/dev/random` , or a hardware device capable of generating random numbers. The `/dev/random` device can block for a long time if there are not enough events going on to generate sufficient entropy.

Compliant Solution (Windows)

On Windows platforms, the `BcryptGenRandom()` function can be used to generate cryptographically strong random numbers. The Microsoft Developer Network `BCryptGenRandom()` reference [MSDN] states:

> The default random number provider implements an algorithm for generating random numbers that complies with the NIST SP800-90 standard, specifically the CTR_DRBG portion of that standard.

```
#include <Windows.h>
#include <bcrypt.h>
#include <stdio.h>

#pragma comment(lib, "Bcrypt")

void func( void ) {
   BCRYPT_ALG_HANDLE Prov;
   int Buffer;
   if (!BCRYPT_SUCCESS(
           BCryptOpenAlgorithmProvider(&Prov, BCRYPT_RNG_ALGORITHM,
                                       NULL, 0))) {
     /* handle error */
   }
   if (!BCRYPT_SUCCESS(BCryptGenRandom(Prov, ( PUCHAR ) (&Buffer),
                                       sizeof (Buffer), 0))) {
     /* handle error */
   }
   printf ( "Random number: %d\n" , Buffer);
   BCryptCloseAlgorithmProvider(Prov, 0);
}
```

Rule MSC33-C

Synopsis: Do not pass invalid data to the asctime() function
Language: C++
Level: 1
Category: Security

Description

The C Standard, 7.27.3.1 [ISO/IEC 9899:2011], provides the following sample implementation of the `asctime()` function:

```
char *asctime( const struct tm *timeptr) {
   static const char wday_name[ 7 ][ 3 ] = {
```

```
    "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
  };
  static const char mon_name[ 12 ][ 3 ] = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
  };
  static char result[ 26 ];
  sprintf(
    result,
    "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
    wday_name[timeptr->tm_wday],
    mon_name[timeptr->tm_mon],
    timeptr->tm_mday, timeptr->tm_hour,
    timeptr->tm_min, timeptr->tm_sec,
    1900 + timeptr->tm_year
  );
  return result;
}
```

This function is supposed to output a character string of 26 characters at most, including the terminating null character. If we count the length indicated by the format directives, we arrive at 25. Taking into account the terminating null character, the array size of the string appears sufficient.

However, this implementation assumes that the values of the `struct tm` data are within normal ranges and does nothing to enforce the range limit. If any of the values print more characters than expected, the `sprintf()` function may overflow the `result` array. For example, if `tm_year` has the value `12345`, then 27 characters (including the terminating null character) are printed, resulting in a buffer overflow.

The *POSIX® Base Specifications* [IEEE Std 1003.1:2013] says the following about the `asctime()` and `asctime_r()` functions:

> These functions are included only for compatibility with older implementations. They have undefined behavior if the resulting string would be too long, so the use of these functions should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffers in such a way as to cause applications to fail, or possible system security violations. Also, these functions do not support localized date and time formats. To avoid these problems, applications should use `strftime()` to generate strings from broken-down times.

The C Standard, Annex K, also defines `asctime_s()` , which can be used as a secure substitute for `asctime()` .

The `asctime()` function appears in the list of obsolescent functions in MSC24-C. Do not use deprecated or obsolescent functions.

Noncompliant Code Example

This noncompliant code example invokes the `asctime()` function with potentially unsanitized data:

```
#include <time.h>

void func( struct tm *time_tm) {
  char * time = asctime (time_tm);
  /* ... */
}
```

Compliant Solution (`strftime()`)

The `strftime()` function allows the programmer to specify a more rigorous format and also to specify the maximum size of the resulting time string:

```
#include <time.h>

enum { maxsize = 26 };

void func( struct tm * time ) {
  char s[maxsize];
  /* Current time representation for locale */
  const char *format = "%c" ;

  size_t size = strftime (s, maxsize, format, time );
}
```

This call has the same effects as `asctime()` but also ensures that no more than `maxsize` characters are printed, preventing buffer overflow.

Compliant Solution (`asctime_s()`)

The C Standard, Annex K, defines the `asctime_s()` function, which serves as a close replacement for the `asctime()` function but requires an additional argument that specifies the maximum size of the resulting time string:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>

enum { maxsize = 26 };

void func( struct tm *time_tm) {
  char buffer[maxsize];

  if (asctime_s(buffer, maxsize, &time_tm)) {
    /* Handle error */
  }
}
```

Rule MSC51-CPP

Synopsis: Ensure your random number generator is properly seeded
Language: C++
Level: 1
Category: Security

Description

A pseudorandom number generator (PRNG) is a deterministic algorithm capable of generating sequences of numbers that approximate the properties of random numbers. Each sequence is completely determined by the initial state of the PRNG and the algorithm for changing the state. Most PRNGs make it possible to set the initial state, also called the *seed state*. Setting the initial state is called *seeding* the PRNG.

Calling a PRNG in the same initial state, either without seeding it explicitly or by seeding it with a constant value, results in generating the same sequence of random numbers in different runs of the program. Consider a PRNG function that is seeded with some initial seed value and is consecutively called to produce a sequence of random numbers. If the PRNG is subsequently seeded with the same initial seed value, then it will generate the same sequence.

Consequently, after the first run of an improperly seeded PRNG, an attacker can predict the sequence of random numbers that will be generated in the future runs. Improperly seeding or failing to seed the PRNG can lead to vulnerabilities, especially in security protocols.

The solution is to ensure that a PRNG is always properly seeded with an initial seed value that will not be predictable or controllable by an attacker. A properly seeded PRNG will generate a different sequence of random numbers each time it is run.

Not all random number generators can be seeded. True random number generators that rely on hardware to produce completely unpredictable results do not need to be and cannot be seeded. Some high-quality PRNGs, such as the `/dev/random` device on some UNIX systems, also cannot be seeded. This rule applies only to algorithmic PRNGs that can be seeded.

Noncompliant Code Example

This noncompliant code example generates a sequence of 10 pseudorandom numbers using the Mersenne Twister engine. No matter how many times this code is executed, it always produces the same sequence because the default seed is used for the engine.

```
#include <random>
#include <iostream>

void f() {
  std::mt19937 engine;

  for ( int i = 0; i < 10; ++i) {
    std::cout << engine() << ", " ;
  }
}
```

The output of this example follows.

```
1st run: 3499211612 , 581869302 , 3890346734 , 3586334585 , 545404204 ,
4161255391 , 3922919429 , 949333985 , 2715962298 , 1323567403 ,
2nd run: 3499211612 , 581869302 , 3890346734 , 3586334585 , 545404204 ,
4161255391 , 3922919429 , 949333985 , 2715962298 , 1323567403 ,
...
nth run: 3499211612 , 581869302 , 3890346734 , 3586334585 , 545404204 ,
4161255391 , 3922919429 , 949333985 , 2715962298 , 1323567403 ,
```

Noncompliant Code Example

This noncompliant code example improves the previous noncompliant code example by seeding the random number generation engine with the current time. However, this approach is still unsuitable when an attacker can control the time at which the seeding is executed. Predictable seed values can result in exploits when the subverted PRNG is used.

```
#include <ctime>
#include <random>
```

```
#include <iostream>

void f() {
  std::time_t t;
  std::mt19937 engine(std::time(&t));

  for(int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

Compliant Solution

This compliant solution uses `std::random_device` to generate a random value for seeding the Mersenne Twister engine object. The values generated by `std::random_device` are nondeterministic random numbers when possible, relying on random number generation devices, such as `/dev/random`. When such a device is not available, `std::random_device` may employ a random number engine; however, the initial value generated should have sufficient randomness to serve as a seed value.

```
#include <random>
#include <iostream>

void f() {
  std::random_device dev;
  std::mt19937 engine(dev());

  for(int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

The output of this example follows.

```
1st run: 3921124303, 1253168518, 1183339582, 197772533, 83186419,
2599073270, 3238222340, 101548389, 296330365, 3335314032,
2nd run: 2392369099, 2509898672, 2135685437, 3733236524, 883966369,
2529945396, 764222328, 138530885, 4209173263, 1693483251,
3rd run: 914243768, 2191798381, 2961426773, 3791073717, 2222867426,
1092675429, 2202201605, 850375565, 3622398137, 422940882,
...
```

Rule STR31-C

Synopsis: Guarantee that storage for strings has sufficient space for character data and the null terminator
Language: C++
Level: 1
Category: Security

Description

Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013b]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character. (See STR03-C. Do not inadvertently

truncate a string.)

When strings live on the heap, this rule is a specific instance of MEM35-C. Allocate sufficient memory for an object. Because strings are represented as arrays of characters, this rule is related to both ARR30-C. Do not form or use out-of-bounds pointers or array subscripts and ARR38-C. Guarantee that library functions do not form invalid pointers.

Noncompliant Code Example (Off-by-One Error)

This noncompliant code example demonstrates an *off-by-one* error [Dowd 2006]. The loop copies data from `src` to `dest` . However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of `dest` .

```
#include <stddef.h>

void copy( size_t n, char src[n], char dest[n]) {
    size_t i;

    for (i = 0; src[i] && (i < n); ++i) {
      dest[i] = src[i];
    }
    dest[i] = '\0' ;
}
```

Compliant Solution (Off-by-One Error)

In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to `dest` :

```
#include <stddef.h>

void copy( size_t n, char src[n], char dest[n]) {
    size_t i;

    for (i = 0; src[i] && (i < n - 1); ++i) {
      dest[i] = src[i];
    }
    dest[i] = '\0' ;
}
```

Noncompliant Code Example (`gets()`)

The `gets()` function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from `stdin` . This noncompliant code example assumes that `gets()` will not read more than `BUFFER_SIZE - 1` characters from `stdin` . This is an invalid assumption, and the resulting operation can result in a buffer overflow.

The `gets()` function reads characters from the `stdin` into a destination array until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array.

```
#include <stdio.h>
```

```
#define BUFFER_SIZE 1024

void func( void ) {
   char buf[BUFFER_SIZE];
   if ( gets (buf) == NULL) {
     /* Handle error */
   }
}
```

See also MSC24-C. Do not use deprecated or obsolescent functions.

Compliant Solution (`fgets()`)

The `fgets()` function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from `stdin` to `buf` cannot exceed the allocated memory:

```
#include <stdio.h>
#include <string.h>

enum { BUFFERSIZE = 32 };

void func( void ) {
   char buf[BUFFERSIZE];
   int ch;

   if ( fgets (buf, sizeof (buf), stdin)) {
     /* fgets() succeeded; scan for newline character */
     char *p = strchr (buf, '\n' );
     if (p) {
       *p = '\0' ;
     } else {
       /* Newline not found; flush stdin to end of line */
       while ((ch = getchar ()) != '\n' && ch != EOF)
         ;
       if (ch == EOF && ! feof (stdin) && ! ferror (stdin)) {
           /* Character resembles EOF; handle error */
       }
     }
   } else {
     /* fgets() failed; handle error */
   }
}
```

The `fgets()` function is not a strict replacement for the `gets()` function because `fgets()` retains the newline character (if read) and may also return a partial line. It is possible to use `fgets()` to safely process input lines too long to store in the destination array, but this is not recommended for performance reasons. Consider using one of the following compliant solutions when replacing `gets()`.

Compliant Solution (`gets_s()`)

The `gets_s()` function reads, at most, one less than the number of characters specified from the stream pointed to by `stdin` into an array.

The C Standard, Annex K [ISO/IEC 9899:2011], states

> No additional characters are read after a new-line character (which is discarded) or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read into the destination array, or if a read error occurs during the operation, then the first character in the destination array is set to the null character and the other elements of the array take unspecified values:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>

enum { BUFFERSIZE = 32 };

void func( void ) {
  char buf[BUFFERSIZE];

  if (gets_s(buf, sizeof (buf)) == NULL) {
    /* Handle error */
  }
}
```

Compliant Solution (`getline()` , POSIX)

The `getline()` function is similar to the `fgets()` function but can dynamically allocate memory for the input buffer. If passed a null pointer, `getline()` dynamically allocates a buffer of sufficient size to hold the input. If passed a pointer to dynamically allocated storage that is too small to hold the contents of the string, the `getline()` function resizes the buffer, using `realloc()` , rather than truncating the input. If successful, the `getline()` function returns the number of characters read, which can be used to determine if the input has any null characters before the newline. The `getline()` function works only with dynamically allocated buffers. Allocated memory must be explicitly deallocated by the caller to avoid memory leaks. (See MEM31-C. Free dynamically allocated memory when no longer needed.)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func( void ) {
  int ch;
  size_t buffer_size = 32;
  char *buffer = malloc (buffer_size);

  if (!buffer) {
    /* Handle error */
    return ;
  }

  if ((ssize_t size = getline(&buffer, &buffer_size, stdin))
       == -1) {
    /* Handle error */
  } else {
    char *p = strchr (buffer, '\n' );
    if (p) {
      *p = '\0' ;
    } else {
      /* Newline not found; flush stdin to end of line */
```

```
      while ((ch = getchar()) != '\n' && ch != EOF)
        ;
      if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
         /* Character resembles EOF; handle error */
      }
    }
  }
  free(buffer);
}
```

Note that the `getline()` function uses an in-band error indicator, in violation of ERR02-C. Avoid in-band error indicators.

Noncompliant Code Example (`getchar()`)

Reading one character at a time provides more flexibility in controlling behavior, though with additional performance overhead. This noncompliant code example uses the `getchar()` function to read one character at a time from `stdin` instead of reading the entire line at once. The `stdin` stream is read until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. Similar to the noncompliant code example that invokes `gets()` , there are no guarantees that this code will not result in a buffer overflow.

```
#include <stdio.h>

enum { BUFFERSIZE = 32 };

void func( void ) {
  char buf[BUFFERSIZE];
  char *p;
  int ch;
  p = buf;
  while ((ch = getchar()) != '\n' && ch != EOF) {
    *p++ = ( char )ch;
  }
  *p++ = 0;
  if (ch == EOF) {
      /* Handle EOF or error */
  }
}
```

After the loop ends, if `ch == EOF` , the loop has read through to the end of the stream without encountering a newline character, or a read error occurred before the loop encountered a newline character. To conform to FIO34-C. Distinguish between characters read from a file and EOF or WEOF, the error-handling code must verify that an end-of-file or error has occurred by calling `feof()` or `ferror()` .

Compliant Solution (`getchar()`)

In this compliant solution, characters are no longer copied to `buf` once `index == BUFFERSIZE - 1` , leaving room to null-terminate the string. The loop continues to read characters until the end of the line, the end of the file, or an error is encountered. When `chars_read > index` , the input string has been truncated.

```
#include <stdio.h>
```

```
enum { BUFFERSIZE = 32 };

void func( void ) {
  char buf[BUFFERSIZE];
  int ch;
  size_t index = 0;
  size_t chars_read = 0;

  while ((ch = getchar ()) != '\n' && ch != EOF) {
    if (index < sizeof (buf) - 1) {
      buf[index++] = ( char )ch;
    }
    chars_read++;
  }
  buf[index] = '\0' ;  /* Terminate string */
  if (ch == EOF) {
    /* Handle EOF or error */
  }
  if (chars_read > index) {
    /* Handle truncation */
  }
}
```

Noncompliant Code Example (`fscanf( )`)

In this noncompliant example, the call to `fscanf( )` can result in a write outside the character array `buf` :

```
#include <stdio.h>

enum { BUF_LENGTH = 1024 };

void get_data( void ) {
  char buf[BUF_LENGTH];
  if (1 != fscanf (stdin, "%s" , buf)) {
    /* Handle error */
  }

  /* Rest of function */
}
```

Compliant Solution (`fscanf( )`)

In this compliant solution, the call to `fscanf( )` is constrained not to overflow `buf` :

```
#include <stdio.h>

enum { BUF_LENGTH = 1024 };

void get_data( void ) {
  char buf[BUF_LENGTH];
  if (1 != fscanf (stdin, "%1023s" , buf)) {
    /* Handle error */
  }
```

```
  /* Rest of function */
}
```

Noncompliant Code Example (`argv`)

In a hosted environment, arguments read from the command line are stored in process memory. The function `main()` , called at program startup, is typically declared as follows when the program accepts command-line arguments:

```
int main( int argc, char *argv[]) { /* ... */ }
```

Command-line arguments are passed to `main()` as pointers to strings in the array members `argv[0]` through `argv[argc - 1]` . If the value of `argc` is greater than 0, the string pointed to by `argv[0]` is, by convention, the program name. If the value of `argc` is greater than 1, the strings referenced by `argv[1]` through `argv[argc - 1]` are the program arguments.

Vulnerabilities can occur when inadequate space is allocated to copy a command-line argument or other program input. In this noncompliant code example, an attacker can manipulate the contents of `argv[0]` to cause a buffer overflow:

```
#include <string.h>

int main( int argc, char *argv[]) {
  /* Ensure argv[0] is not null */
  const char * const name = (argc && argv[0]) ? argv[0] : "" ;
  char prog_name[128];
  strcpy (prog_name, name);

  return 0;
}
```

Compliant Solution (`argv`)

The `strlen()` function can be used to determine the length of the strings referenced by `argv[0]` through `argv[argc - 1]` so that adequate memory can be dynamically allocated.

```
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[]) {
  /* Ensure argv[0] is not null */
  const char * const name = (argc && argv[0]) ? argv[0] : "" ;
  char *prog_name = ( char *) malloc ( strlen (name) + 1);
  if (prog_name != NULL) {
    strcpy (prog_name, name);
  } else {
    /* Handle error */
  }
  free (prog_name);
  return 0;
}
```

Remember to add a byte to the destination string size to accommodate the null-termination character.

Compliant Solution (`argv`)

The `strcpy_s()` function provides additional safeguards, including accepting the size of the destination buffer as an additional argument. (See STR07-C. Use the bounds-checking interfaces for string manipulation.)

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[]) {
   /* Ensure argv[0] is not null */
   const char * const name = (argc && argv[0]) ? argv[0] : "" ;
   char *prog_name;
   size_t prog_size;

   prog_size = strlen (name) + 1;
   prog_name = ( char *) malloc (prog_size);

   if (prog_name != NULL) {
     if (strcpy_s(prog_name, prog_size, name)) {
       /* Handle  error */
     }
   } else {
     /* Handle error */
   }
   /* ... */
   free (prog_name);
   return 0;
}
```

The `strcpy_s()` function can be used to copy data to or from dynamically allocated memory or a statically allocated array. If insufficient space is available, `strcpy_s()` returns an error.

Compliant Solution (`argv`)

If an argument will not be modified or concatenated, there is no reason to make a copy of the string. Not copying a string is the best way to prevent a buffer overflow and is also the most efficient solution. Care must be taken to avoid assuming that `argv[0]` is non-null.

```
int main( int argc, char *argv[]) {
   /* Ensure argv[0] is not null */
   const char * const prog_name = (argc && argv[0]) ? argv[0] : "" ;
   /* ... */
   return 0;
}
```

Noncompliant Code Example (`getenv()`)

According to the C Standard, 7.22.4.6 [ISO/IEC 9899:2011]

> The `getenv` function searches an environment list, provided by the host environment, for a string that matches the string pointed to by `name` . The set of environment names and the method for altering the environment list are implementation defined.

Environment variables can be arbitrarily large, and copying them into fixed-length arrays without first determining the size and allocating adequate storage can result in a buffer overflow.

```
#include <stdlib.h>
#include <string.h>

void func( void ) {
  char buff[256];
  char *editor = getenv( "EDITOR" );
  if (editor == NULL) {
    /* EDITOR environment variable not set */
  } else {
    strcpy(buff, editor);
  }
}
```

Compliant Solution (`getenv()`)

Environmental variables are loaded into process memory when the program is loaded. As a result, the length of these strings can be determined by calling the `strlen()` function, and the resulting length can be used to allocate adequate dynamic memory:

```
#include <stdlib.h>
#include <string.h>

void func( void ) {
  char *buff;
  char *editor = getenv( "EDITOR" );
  if (editor == NULL) {
    /* EDITOR environment variable not set */
  } else {
    size_t len = strlen(editor) + 1;
    buff = (char *) malloc(len);
    if (buff == NULL) {
      /* Handle error */
    }
    memcpy(buff, editor, len);
    free(buff);
  }
}
```

Noncompliant Code Example (`sprintf()`)

In this noncompliant code example, `name` refers to an external string; it could have originated from user input, the file system, or the network. The program constructs a file name from the string in preparation for opening the file.

```
#include <stdio.h>

void func( const char *name) {
  char filename[128];
  sprintf(filename, "%s.txt", name);
}
```

Because the `sprintf()` function makes no guarantees regarding the length of the generated string, a sufficiently long string in `name` could generate a buffer overflow.

Compliant Solution (`sprintf()`)

The buffer overflow in the preceding noncompliant example can be prevented by adding a precision to the `%s` conversion specification. If the precision is specified, no more than that many bytes are written. The precision `123` in this compliant solution ensures that `filename` can contain the first 123 characters of `name` , the `.txt` extension, and the null terminator.

```
#include <stdio.h>

void func( const char *name) {
  char filename[128];
  sprintf (filename, "%.123s.txt" , name);
}
```

Compliant Solution (`snprintf()`)

A more general solution is to use the `snprintf()` function:

```
#include <stdio.h>

void func( const char *name) {
  char filename[128];
  snprintf(filename, sizeof (filename), "%s.txt" , name);
}
```

Rule STR32-C

Synopsis: Do not pass a non-null-terminated character sequence to a library function that expects a string
Language: C++
Level: 1
Category: Security

Description

Many library functions accept a string or wide string argument with the constraint that the string they receive is properly null-terminated. Passing a character sequence or wide character sequence that is not null-terminated to such a function can result in accessing memory that is outside the bounds of the object. Do not pass a character sequence or wide character sequence that is not null-terminated to a library function that expects a string or wide string argument.

Noncompliant Code Example

This code example is noncompliant because the character sequence `c_str` will not be null-terminated when passed as an argument to `printf()`. (See STR11-C. Do not specify the bound of a character array initialized with a string literal on how to properly initialize character arrays.)

```
#include <stdio.h>

void func( void ) {
  char c_str[3] = "abc" ;
  printf ( "%s\n" , c_str);
}
```

Compliant Solution

This compliant solution does not specify the bound of the character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character.

```
#include <stdio.h>

void func( void ) {
   char c_str[] = "abc" ;
   printf ( "%s\n" , c_str);
}
```

Noncompliant Code Example

This code example is noncompliant because the wide character sequence `cur_msg` will not be null-terminated when passed to `wcslen()` . This will occur if `lessen_memory_usage()` is invoked while `cur_msg_size` still has its initial value of 1024.

```
#include <stdlib.h>
#include <wchar.h>

wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;

void lessen_memory_usage( void ) {
   wchar_t *temp;
   size_t temp_size;

   /* ... */

   if (cur_msg != NULL) {
     temp_size = cur_msg_size / 2 + 1;
     temp = realloc (cur_msg, temp_size * sizeof ( wchar_t ));
     /* temp &and cur_msg may no longer be null-terminated */
     if (temp == NULL) {
       /* Handle error */
     }

     cur_msg = temp;
     cur_msg_size = temp_size;
     cur_msg_len = wcslen(cur_msg);
   }
}
```

Compliant Solution

In this compliant solution, `cur_msg` will always be null-terminated when passed to `wcslen()` :

```
#include <stdlib.h>
#include <wchar.h>

wchar_t *cur_msg = NULL;
```

```
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;

void lessen_memory_usage( void ) {
  wchar_t *temp;
  size_t temp_size;

  /* ... */

  if (cur_msg != NULL) {
    temp_size = cur_msg_size / 2 + 1;
    temp = realloc (cur_msg, temp_size * sizeof ( wchar_t ));
    /* temp and cur_msg may no longer be null-terminated */
    if (temp == NULL) {
      /* Handle error */
    }

    cur_msg = temp;
    /* Properly null-terminate cur_msg */
    cur_msg[temp_size - 1] = L '\0' ;
    cur_msg_size = temp_size;
    cur_msg_len = wcslen(cur_msg);
  }
}
```

Noncompliant Code Example (`strncpy( )`)

Although the `strncpy( )` function takes a string as input, it does not guarantee that the resulting value is still null-terminated. In the following noncompliant code example, if no null character is contained in the first `n` characters of the `source` array, the result will not be null-terminated. Passing a non-null-terminated character sequence to `strlen( )` is undefined behavior.

```
#include <string.h>

enum { STR_SIZE = 32 };

size_t func( const char *source) {
  char c_str[STR_SIZE];
  size_t ret = 0;

  if (source) {
    c_str[ sizeof (c_str) - 1] = '\0' ;
    strncpy (c_str, source, sizeof (c_str));
    ret = strlen (c_str);
  } else {
    /* Handle null pointer */
  }
  return ret;
}
```

Compliant Solution (Truncation)

This compliant solution is correct if the programmer's intent is to truncate the string:

```
#include <string.h>
```

```
enum { STR_SIZE = 32 };

size_t func( const char *source) {
  char c_str[STR_SIZE];
  size_t ret = 0;

  if (source) {
    strncpy (c_str, source, sizeof (c_str) - 1);
    c_str[ sizeof (c_str) - 1] = '\0' ;
    ret = strlen (c_str);
  } else {
    /* Handle null pointer */
  }
  return ret;
}
```

Compliant Solution (Truncation, strncpy_s())

The C Standard, Annex K `strncpy_s()` function can also be used to copy with truncation. The `strncpy_s()` function copies up to `n` characters from the source array to a destination array. If no null character was copied from the source array, then the `n` th position in the destination array is set to a null character, guaranteeing that the resulting string is null-terminated.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

enum { STR_SIZE = 32 };

size_t func( const char *source) {
  char a[STR_SIZE];
  size_t ret = 0;

  if (source) {
    errno_t err = strncpy_s(
      a, sizeof (a), source, strlen (source)
    );
    if (err != 0) {
      /* Handle error */
    } else {
      ret = strnlen_s(a, sizeof (a));
    }
  } else {
     /* Handle null pointer */
  }
  return ret;
}
```

Compliant Solution (Copy without Truncation)

If the programmer's intent is to copy without truncation, this compliant solution copies the data and guarantees that the resulting array is null-terminated. If the string cannot be copied, it is handled as an error condition.

```
#include <string.h>
```

```
enum { STR_SIZE = 32 };

size_t func( const char *source) {
  char c_str[STR_SIZE];
  size_t ret = 0;

  if (source) {
    if ( strlen (source) < sizeof (c_str)) {
      strcpy (c_str, source);
      ret = strlen (c_str);
    } else {
      /* Handle string-too-large */
    }
  } else {
    /* Handle null pointer */
  }
  return ret;
}
```

Rule STR38-C

Synopsis: Do not confuse narrow and wide character strings and functions
Language: C++
Level: 1
Category: Security

Description

Passing narrow string arguments to wide string functions or wide string arguments to narrow string functions can lead to unexpected and undefined behavior. Scaling problems are likely because of the difference in size between wide and narrow characters. (See ARR39-C. Do not add or subtract a scaled integer to a pointer.) Because wide strings are terminated by a null wide character and can contain null bytes, determining the length is also problematic.

Because `wchar_t` and `char` are distinct types, many compilers will produce a warning diagnostic if an inappropriate function is used. (See MSC00-C. Compile cleanly at high warning levels.)

Noncompliant Code Example (Wide Strings with Narrow String Functions)

This noncompliant code example incorrectly uses the `strncpy()` function in an attempt to copy up to 10 wide characters. However, because wide characters can contain null bytes, the copy operation may end earlier than anticipated, resulting in the truncation of the wide string.

```
#include <stddef.h>
#include <string.h>

void func( void ) {
  wchar_t wide_str1[]  = L"0123456789";
  wchar_t wide_str2[] =  L"0000000000";
```

```
   strncpy(wide_str2, wide_str1, 10);
}
```

Noncompliant Code Example (Narrow Strings with Wide String Functions)

This noncompliant code example incorrectly invokes the `wcsncpy()` function to copy up to 10 wide characters from `narrow_str1` to `narrow_str2`. Because `narrow_str2` is a narrow string, it has insufficient memory to store the result of the copy and the copy will result in a buffer overflow.

```
#include <wchar.h>

void func( void ) {
   char narrow_str1[] = "01234567890123456789";
   char narrow_str2[] = "0000000000";

   wcsncpy(narrow_str2, narrow_str1, 10);
}
```

Compliant Solution

This compliant solution uses the proper-width functions. Using `wcsncpy()` for wide character strings and `strncpy()` for narrow character strings ensures that data is not truncated and buffer overflow does not occur.

```
#include <string.h>
#include <wchar.h>

void func( void ) {
   wchar_t wide_str1[] = L"0123456789";
   wchar_t wide_str2[] = L"0000000000";
   /* Use of proper-width function */
   wcsncpy(wide_str2, wide_str1, 10);

   char narrow_str1[] = "0123456789";
   char narrow_str2[] = "0000000000";
   /* Use of proper-width function */
   strncpy(narrow_str2, narrow_str1, 10);
}
```

Noncompliant Code Example (`strlen()`)

In this noncompliant code example, the `strlen()` function is used to determine the size of a wide character string:

```
#include <stdlib.h>
#include <string.h>

void func( void ) {
   wchar_t wide_str1[] = L"0123456789";
   wchar_t *wide_str2 = (wchar_t *)malloc( strlen(wide_str1) + 1);
   if (wide_str2 == NULL) {
     /* Handle error */
```

```
  }
  /* ... */
  free(wide_str2);
  wide_str2 = NULL;
}
```

The `strlen()` function determines the number of characters that precede the terminating null character. However, wide characters can contain null bytes, particularly when expressing characters from the ASCII character set, as in this example. As a result, the `strlen()` function will return the number of bytes preceding the first null byte in the wide string.

Compliant Solution

This compliant solution correctly calculates the number of bytes required to contain a copy of the wide string, including the terminating null wide character:

```
#include <stdlib.h>
#include <wchar.h>

void func( void ) {
  wchar_t wide_str1[] = L"0123456789";
  wchar_t *wide_str2 = (wchar_t*) malloc(
    (wcslen(wide_str1) + 1) * sizeof(wchar_t));
  if (wide_str2 == NULL) {
    /* Handle error */
  }
  /* ... */

  free(wide_str2);
  wide_str2 = NULL;
}
```

Rule STR50-CPP

Synopsis: Guarantee that storage for strings has sufficient space for character data and the null terminator
Language: C++
Level: 1
Category: Security

Description

Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ `std::basic_string` template requires no such character.

Noncompliant Code Example

Because the input is unbounded, the following code could lead to a buffer overflow.

```
#include <iostream>
```

```
void f() {
  char buf[12];
  std::cin >> buf;
}
```

Noncompliant Code Example

To solve this problem, it may be tempting to use the `std::ios_base::width()` method, but there still is a trap, as shown in this noncompliant code example.

```
#include <iostream>

void f() {
  char bufOne[12];
  char bufTwo[12];
  std::cin.width(12);
  std::cin >> bufOne;
  std::cin >> bufTwo;
}
```

In this example, the first read will not overflow, but could fill `bufOne` with a truncated string. Furthermore, the second read still could overflow `bufTwo` . The C++ Standard, [istream.extractors], paragraphs 79 [ISO/IEC 14882-2014], describes the behavior of `operator>>(basic_istream &, charT *)` and, in part, states the following:

> `operator>>` then stores a null byte (`charT()`) in the next position, which may be the first position if no characters were extracted. `operator>>` then calls `width(0)` .

Consequently, it is necessary to call `width()` prior to each `operator>>` call passing a bounded array. However, this does not account for the input being truncated, which may lead to information loss or a possible vulnerability.

Compliant Solution

The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use `std::string` instead of a bounded array, as in this compliant solution.

```
#include <iostream>
#include <string>

void f() {
  std::string input;
  std::string stringOne, stringTwo;
  std::cin >> stringOne >> stringTwo;
}
```

Noncompliant Code Example

In this noncompliant example, the unformatted input function `std::basic_istream<T>::read()` is used to read an unformatted character array of 32 characters from the given file. However, the `read()` function does not guarantee that the string will be null terminated, so the subsequent call of the `std::string` constructor results in undefined behavior if the character array does not contain a null terminator.

```
#include <fstream>
#include <string>

void f(std::istream &in) {
  char buffer[32];
  try {
    in.read(buffer, sizeof(buffer));
  } catch (std::ios_base::failure &e) {
    // Handle error
  }

  std::string str(buffer);
  // ...
}
```

Compliant Solution

This compliant solution assumes that the input from the file is at most 32 characters. Instead of inserting a null terminator, it constructs the `std::string` object based on the number of characters read from the input stream. If the size of the input is uncertain, it is better to use `std::basic_istream<T>::readsome()` or a formatted input function, depending on need.

```
#include <fstream>
#include <string>

void f(std::istream &in) {
  char buffer[32];
  try {
    in.read(buffer, sizeof(buffer));
  } catch (std::ios_base::failure &e) {
    // Handle error
  }
  std::string str(buffer, in.gcount());
  // ...
}
```

Rule STR51-CPP

Synopsis: Do not attempt to create a std::string from a null pointer
Language: C++
Level: 1
Category: Security

Description

The `std::basic_string` type uses the *traits* design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Specifically, the `std::basic_string` class is paired with `std::char_traits` to create the `std::string`, `std::wstring`, `std::u16string`, and `std::u32string` classes. The `std::char_traits` class is explicitly specialized to provide policy-based implementation details to the `std::basic_string` type. One such implementation detail is the `std::char_traits::length()` function, which is frequently used to determine the number of characters in a null-terminated string. According to the C++ Standard, [char.traits.require], Table 62

[ISO/IEC 14882-2014], passing a null pointer to this function is undefined behavior because it would result in dereferencing a null pointer.

The following `std::basic_string` member functions result in a call to `std::char_traits::length()`:

- `basic_string::basic_string(const charT *, const Allocator &)`
- `basic_string &basic_string::append(const charT *)`
- `basic_string &basic_string::assign(const charT *)`
- `basic_string &basic_string::insert(size_type, const charT *)`
- `basic_string &basic_string::replace(size_type, size_type, const charT *)`
- `basic_string &basic_string::replace(const_iterator, const_iterator, const charT *)`
- `size_type basic_string::find(const charT *, size_type)`
- `size_type basic_string::rfind(const charT *, size_type)`
- `size_type basic_string::find_first_of(const charT *, size_type)`
- `size_type basic_string::find_last_of(const charT *, size_type)`
- `size_type basic_string::find_first_not_of(const charT *, size_type)`
- `size_type basic_string::find_last_not_of(const charT *, size_type)`
- `int basic_string::compare(const charT *)`
- `int basic_string::compare(size_type, size_type, const charT *)`
- `basic_string &basic_string::operator=(const charT *)`
- `basic_string &basic_string::operator+=(const charT *)`

The following `std::basic_string` nonmember functions result in a call to to `std::char_traits::length()`:

- `basic_string operator+(const charT *, const basic_string&)`
- `basic_string operator+(const charT *, basic_string &&)`
- `basic_string operator+(const basic_string &, const charT *)`
- `basic_string operator+(basic_string &&, const charT *)`
- `bool operator==(const charT *, const basic_string &)`
- `bool operator==(const basic_string &, const charT *)`
- `bool operator!=(const charT *, const basic_string &)`
- `bool operator!=(const basic_string &, const charT *)`
- `bool operator<(const charT *, const basic_string &)`
- `bool operator<(const basic_string &, const charT *)`
- `bool operator>(const charT *, const basic_string &)`
- `bool operator>(const basic_string &, const charT *)`
- `bool operator<=(const charT *, const basic_string &)`
- `bool operator<=(const basic_string &, const charT *)`
- `bool operator>=(const charT *, const basic_string &)`
- `bool operator>=(const basic_string &, const charT *)`

Do not call any of the preceding functions with a null pointer as the `const charT *` argument.

This rule is a specific instance of EXP34-C. Do not dereference null pointers.

Implementation Details

Some standard library vendors, such as libstdc++, throw a `std::logic_error` when a null pointer is used in the above function calls, though not when calling `std::char_traits::length()`.

However, `std::logic_error` is not a requirement of the C++ Standard, and some vendors (e.g., libc++ and the Microsoft Visual Studio STL) do not implement this behavior. For portability, you should not rely on this behavior.

Noncompliant Code Example

In this noncompliant code example, a `std::string` object is created from the results of a call to `std::getenv()`. However, because `std::getenv()` returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs).

```
#include <cstdlib>
#include <string>

void f() {
  std::string tmp(std::getenv("TMP"));
  if (!tmp.empty()) {
    // ...
  }
}
```

Compliant Solution

In this compliant solution, the results from the call to `std::getenv()` are checked for null before the `std::string` object is constructed.

```
#include <cstdlib>
#include <string>

void f() {
  const char *tmpPtrVal = std::getenv("TMP");
  std::string tmp(tmpPtrVal ? tmpPtrVal : "");
  if (!tmp.empty()) {
    // ...
  }
}
```

Static Objects

This chapter concerns static objects. Static objects are global objects, static data members, file scope objects and local variables declared static.

Rules

STA#001	If a variable is not intended to change, make it a constant
STA#002	Objects with static storage duration should be declared only within the scope of a class, function, or unnamed namespace

Rule STA#001

Synopsis: If a variable is not intended to change, make it a constant
Language: C++
Level: 9
Category: Static Objects

Description

This way the compiler assures that the variable cannot be changed.

Rule STA#002

Synopsis: Objects with static storage duration should be declared only within the scope of a class, function, or unnamed namespace
Language: C++
Level: 4
Category: Static Objects

Description

This means that declarations of global objects with external linkage are best avoided. Otherwise, such external declarations are only allowed in header files.

Apart from the well known problems related to the use of global data, the order in which constructors of global objects are called at program startup is only specified within each translation unit, but not across different translation units (say, source files). This can be problematic in case the construction of a global object depends on the construction of another global object in a different file. See further the ISC++ explanation.

Note that the term "unnamed namespace" is preferred to "anonymous namespace".

Classes defined outside any namespace or block should be avoided where possible, and prefixed otherwise. The following example (using condensed style for brevity) shows undefined behaviour if the resulting objects are linked together.

```
// file1.cpp
#include <iostream>
// A simple class definition at "file scope".
```

```
// This does not hurt anyone, does it?
    class Demo
    {
    public:
         Demo(int m) : v(m)
             { std::cout << "construct " << v << std::endl; }
         ~Demo()
             { std::cout << "destroyed " << v << std::endl; }
    private:
        int v;
    };

int main (int argc, char *argv[])
{
    Demo dd(7);
    extern void badPractice();  // should come from a header...
    badPractice();
    std::cout << "done ----------" << std::endl;
}

// file2.cpp
#include <iostream>
// Another class definition at "file scope".
class Demo
{
public:
     Demo(double m) : v(m)
         { std::cout << "# construct " << v << std::endl; }
     ~Demo()
         { std::cout << "# destroyed " << v << std::endl; }
private:
    double v;
};

// a declaration of this function should go into a header
void badPractice()
{
    Demo foo(3.14);     // foo has no linkage
    std::cout << "# done ----------" << std::endl;
}
```

In case compilers do not support the "namespace" concept yet, objects at file scope may also be declared with the "static" keyword.

Code Organization

This chapter concerns the organization of code in files. Guidelines for organizing the code makes it easier to find code and can be used to enforce a correct implementation of the system's architecture.

Rules

ORG#001	Enclose all code in header files within include guards
ORG#002	Each file shall be self-contained
ORG#003	From a source file include only header files
ORG#004	Classes that are only accessed via pointers (*) or references (&) shall not be included as header files
ORG#005	Each file shall directly include each header file upon which declarations it directly depends
ORG#006	C++ header files have the extension .h, .hpp or .hxx
ORG#007	C++ source files have the extension .cpp
ORG#009	Avoid unnecessary inclusion
ORG#010	Do not let assertions change the state of the program
ORG#011	Everything must reside in a namespace
ORG#012	Never put "using namespace" in header files
ORG#013	Don't put definitions in header files

Rule ORG#001

Synopsis: Enclose all code in header files within include guards
Language: C++
Level: 3
Category: Code Organization

Description

This minimizes build time and prevents compilation problems due to redefinitions.

Alternative 1: include guards:

Adding the following preprocessor statements at the beginning of the header file prevents multiple inclusion:

```
#ifndef FILENAME_H
#define FILENAME_H
```

And the following statement at the end:

```
#endif // FILENAME_H
```

See [STY#024] for the naming of these guards.

Alternative 2: #pragma once:

Although most modern compilers support this compiler directive, "#pragma once" is not an ISO standard and thus not portable. It's nevertheless allowed to use "pragma once" because

- it never leads to name clashes
- it's less susceptible to typos

```
#pragma once
```

Rule ORG#002

Synopsis: Each file shall be self-contained
Language: C++
Level: 3
Category: Code Organization

Description

This rule means that a file should be compilable in itself. For header files this is not enforced by the compiler. So if the file is a header file, including this file in an empty source file should not lead to compiler errors.

Rule ORG#003

Synopsis: From a source file include only header files
Language: C++
Level: 3
Category: Code Organization

Description

A source file only #includes header files. Source files are **never** #included.

Rule ORG#004

Synopsis: Classes that are only accessed via pointers (*) or references (&) shall not be included as header files
Language: C++
Level: 8
Category: Code Organization

Description

When a class contains pointers or references to other classes, the header files of these referenced classes need not be included in the header file defining the containing class. The implementation file belonging to the containing class does of course need the header file with the referenced classes. By doing so, code using the containing class does not depend on the definition of the referenced classes. This limits dependencies and speeds up the building process.

The here-described usage is explained with an example. Suppose that class *CDemo*, defined in header file *Demo.h*, contains a pointer to Class *CUsed* and a reference to class *COther* defined in files Used.h and

Other.h. Then file Demo.h would look like:

Demo.h

```
class CUsed;
class COther;

class CDemo
{
public:
    ...

protected:
    ...

private:
    CUsed*  m_pUsed;
    COther& m_rOther;
};
```

Since the implementation file of class *CDemo* makes use of classes *CUsed* and *COther*, this file does need to include the corresponding header files and thus the first part of this implementation file looks like:

Demo.cpp

```
#include "......"
#include "Used.h"
#include "Other.h"
#include "Demo.h"
...
```

Note that this rule also holds for function results of class type.

Rule ORG#005

Synopsis: Each file shall directly include each header file upon which declarations it directly depends
Language: C++
Level: 3
Category: Code Organization

Description

A file shall not depend on declarations that are only indirectly included by means of a nested #include directive in some other included header file. Otherwise, it would be unnecessarily difficult to change dependencies of a header file, even if it already was self-contained. It would also be somewhat less obvious which files are the actual users of specific headers.

Exception:an exception to this is the inclusion of "stdafx.h" in Visual Studio. In this header file a lot of files are combined to speed up the compilation process.

Rule ORG#006

Synopsis: C++ header files have the extension .h, .hpp or .hxx

Language: C++
Level: 8
Category: Code Organization

Description

Using other extensions introduce complexity when used in the Visual C++ environment. For ease of use the generally accepted extension .h, .hpp and .hxx are used.

Rule ORG#007

Synopsis: C++ source files have the extension .cpp
Language: C++
Level: 8
Category: Code Organization

Description

C++ source files have the extension .cpp. This makes it easier to understand the meaning of files in a software archive. Allowing other file extensions such as .tpl (templates) and .inl (inline files) doesn't add sufficient value.

Rule ORG#009

Synopsis: Avoid unnecessary inclusion
Language: C++
Level: 6
Category: Code Organization

Description

Do not include any header files that are not used by the including file.

Consider to use "compilation firewalls" such as abstract base classes and the so-called "pimpl" programming idiom to reduce unnecessary compile-time coupling (see [ORG#004]).

Consider to use forward declarations of incomplete types. These declarations shall then be localized in the "__fwd.h" header of a certain module. This not only prevents the occurrence of such type declarations in headers with the wrong basename, but also makes such dependencies much more explicit and traceable by means of an #include directive. Furthermore, the localization of forward declarations in "__fwd.h" headers makes it possible to hide whether a class name is a class template instantiation or not.

Rule ORG#010

Synopsis: Do not let assertions change the state of the program
Language: C++
Level: 1

Category: Code Organization

Description

Assertions are used to verify assumptions made during development. For performance reasons it should be possible to remove assert statements from the compiled code by means of conditional compilation. Of course, when state changing statements are removed this way, the program does not work anymore the way it was intended. For example:

```
// Wrong: allocation not done in release code
char *buf;
assert(buf = (char *) calloc( 20, sizeof(char)));
strcpy(buf, "Hello, World");
free(buf);

// Correct: allocation always done
char *buf;
buf = (char *) calloc( 20, sizeof(char));
assert(buf != nullptr);
strcpy(buf, "Hello, World");
free(buf);
```

Rule ORG#011

Synopsis: Everything must reside in a namespace
Language: C++
Level: 7
Category: Code Organization

Description

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Rule ORG#012

Synopsis: Never put "using namespace" in header files
Language: C++
Level: 5
Category: Code Organization

Description

Suppose you are using two libraries called Foo and Bar:

```
using namespace foo;
using namespace bar;
```

Everything works fine, you can call Blah() from Foo and Quux() from Bar without problems. But one day you upgrade to a new version of Foo 2.0, which now offers a function called Quux(). Now you've got a conflict: Both Foo 2.0 and Bar import Quux() into your global namespace. This is going to take some

effort to fix, especially if the function parameters happen to match.

If you have used foo::Blah() and bar::Quux() then the introduction of foo::Quux() would have been a non-event.

Rule ORG#013

Synopsis: Don't put definitions in header files
Language: C++
Level: 5
Category: Code Organization

Description

Header files are meant as an interface. Definitions are about the implementation of a language entity. The way something is implemented should not be revealed to the outside world. Header files serve both as an easy-to-study interface as well as being open to straight-forward re-implementation.

Exceptions to this rule are constants, enumeration and template definitions.

Portability

This chapter concerns the ability of code to be transferred from one environment to another. Some aspects of the environment may be: the operating system, the hardware platform, the compiler, the GUI, the user's language and localization.

Rules

POR#001	Never use absolute file paths
POR#002	Do not assume that an int and a long have the same size
POR#003	Do not assume that a char is signed or unsigned
POR#004	Do not cast a pointer to a shorter quantity to a pointer to a longer quantity
POR#005	Do not assume that pointers and integers have the same size
POR#006	Use explicit type conversions for arithmetic using signed and unsigned values
POR#007	Do not assume that you know how an instance of a data type is represented in memory
POR#008	Do not depend on underflow or overflow functioning in any special way
POR#009	Do not assume that longs, floats, doubles or long doubles may begin at arbitrary addresses
POR#010	Do not assume that the operands in an expression are evaluated in a definite order, unless the order is specified in the language
POR#011	Do not assume that you know how the invocation mechanism for a method or function is implemented
POR#012	Do not assume that static objects are initialized in any special order when they are spread over multiple source files
POR#014	Do not assume that a short is 16-bits
POR#015	Do not assume that a long is 32-bits
POR#016	Do not assume that the size of an enumeration is the same for all platforms
POR#017	Do not depend on undefined, unspecified, or implementation-defined parts of the language
POR#018	Avoid the use of #pragma directives
POR#019	Header file names shall always be treated as case-sensitive
POR#020	Do not make assumptions about the exact size or layout in memory of an object
POR#021	Avoid the use of conditional compilation
POR#022	Make sure all conversions from a value of one type to another of a narrower type do not slice off significant data
POR#025	Floating point values shall not be compared using the == or != operators
POR#028	Always return a value from main
POR#029	Do not depend on the order of evaluation of arguments to a function
POR#030	Both operands of the remainder operator shall be positive
POR#031	Do not depend on implementation defined behavior of shift operators for built-in types
POR#032	Use nullptr instead of 0 or NULL for a null pointer
POR#033	Do not make assumptions on the size of int
POR#037	Avoid the use of #pragma warning directive.
POR#038	Make sure that the sizes and types of functions are well-defined if exported to other languages

Rule POR#001

Synopsis: Never use absolute file paths
Language: C++
Level: 3
Category: Portability

Rule POR#002

Synopsis: Do not assume that an int and a long have the same size
Language: C++
Level: 4
Category: Portability

Description

It depends on the compiler and platform used if this is the case.

Rule POR#003

Synopsis: Do not assume that a char is signed or unsigned
Language: C++
Level: 4
Category: Portability

Description

In the definition of the C++ language, it has not yet been decided if a *char* is *signed* or *unsigned*. This decision has instead been left to each compiler manufacturer. If this is forgotten and this characteristic is exploited in one way or another, some difficult bugs may appear in the program when another compiler is used. Use *signed char* when you want a one-byte signed numeric type, and use *unsigned char* when you want a one-byte unsigned numeric type. Use plain old *char* when you want to hold characters.

Rule POR#004

Synopsis: Do not cast a pointer to a shorter quantity to a pointer to a longer quantity
Language: C++
Level: 1
Category: Portability

Description

A processor architecture often forbids data of a given size to be allocated at an arbitrary address. For example, a word must begin on an "even" address for MC680x0. If there is a pointer to a *char* which is located at an "odd" address, a type conversion from this *char* pointer to an *int* pointer will cause the program to crash when the *int* pointer is used, since this violates the processor's rules for alignment of data. See also [CON#001].

Rule POR#005

Synopsis: Do not assume that pointers and integers have the same size
Language: C++
Level: 1
Category: Portability

Description

It depends on the compiler and platform used if this is the case.

Rule POR#006

Synopsis: Use explicit type conversions for arithmetic using signed and unsigned values
Language: C++
Level: 5
Category: Portability

Description

This way problems with the representation of *signed* and *unsigned* values in memory are avoided.

Rule POR#007

Synopsis: Do not assume that you know how an instance of a data type is represented in memory
Language: C++
Level: 3
Category: Portability

Description

The representation of data types in memory is highly machine-dependent. By allocating data members to certain addresses, a processor may execute code more efficiently. Because of this, the data structure that represents a class will sometime include holes and be stored differently in different process architectures. Code that depends on a specific representation is, of course, not portable.

Rule POR#008

Synopsis: Do not depend on underflow or overflow functioning in any special way
Language: C++
Level: 4
Category: Portability

Description

This depends on the compiler and platform used.

Rule POR#009

Synopsis: Do not assume that longs, floats, doubles or long doubles may begin at arbitrary addresses
Language: C++
Level: 1
Category: Portability

Description

This depends on the compiler and platform used.

Rule POR#010

Synopsis: Do not assume that the operands in an expression are evaluated in a definite order, unless the order is specified in the language
Language: C++
Level: 1
Category: Portability

Description

If a value is modified twice in the same expression, the result of the expression is undefined except when the order of evaluation is guaranteed for the operators that are used. It is allowed to use knowledge about the evaluation order of boolean expressions since this is specified in the C++ language definition.

Rule POR#011

Synopsis: Do not assume that you know how the invocation mechanism for a method or function is implemented
Language: C++
Level: 4
Category: Portability

Description

This depends on the compiler and platform used.

Rule POR#012

Synopsis:

Do not assume that static objects are initialized in any special order when they are spread over multiple source files

Language: C++
Level: 1
Category: Portability

Description

It is specified in C++ that static objects declared within one source file are initialized in the order they are noted in that file. Usage of this knowledge is acceptable. However when multiple source files are used to compose a module, the initialization order of static objects spread over multiple source files is unpredictable. To solve this problem, it is acceptable when the initialization order needs to be known to place related static objects in one source file.

Rule POR#014

Synopsis: Do not assume that a short is 16-bits
Language: C++
Level: 6
Category: Portability

Description

According to [Stroustrup], the size of a *short* is defined as smaller or equal than the size of an int, but at least 16-bits. This means that the size of a *short* can be more than 16-bits. The size of the Windows types *SHORT* and *USHORT* is defined as 16-bits.

Rule POR#015

Synopsis: Do not assume that a long is 32-bits
Language: C++
Level: 6
Category: Portability

Description

According to [Stroustrup], the size of a *long* is defined as greater or equal than the size of an *int*, but at least 32-bits. This means that the size of a *long* can be more than 32-bits. The size of the Windows types *LONG* and *ULONG* is defined as 32-bits.

Rule POR#016

Synopsis: Do not assume that the size of an enumeration is the same for all platforms
Language: C++
Level: 5
Category: Portability

Description

According to [Stroustrup] the size of an enumeration is the size of some integral type that can hold its range and not larger than the size of an *int*. For instance, if the size of an *int* is 4, the size of an enumeration can be 1, 2 or 4.

Rule POR#017

Synopsis: Do not depend on undefined, unspecified, or implementation-defined parts of the language
Language: C++
Level: 1
Category: Portability

Rule POR#018

Synopsis: Avoid the use of #pragma directives
Language: C++
Level: 6
Category: Portability

Description

Exception 1:
The use of the #pragma preprocessor directive is allowed in specific files which special purpose is to shield or implement platform dependent code. Such a #pragma shall be conditionally defined based on a macro that identifies the platform. This makes the platform dependency explicit. See also [POR#021].

Exception 2: #pragma once is allowed to prevent multiple inclusion of a file. See also [ORG#001].

Rule POR#019

Synopsis: Header file names shall always be treated as case-sensitive
Language: C++
Level: 5
Category: Portability

Rule POR#020

Synopsis: Do not make assumptions about the exact size or layout in memory of an object
Language: C++
Level: 2
Category: Portability

Description

As explained in ISC++, do not depend on specific size or layout assumptions. However, that does not mean that object sizes are completely irrelevant: see also [OLC#011].

Rule POR#021

Synopsis: Avoid the use of conditional compilation
Language: C++
Level: 6
Category: Portability

Description

Conditional compilation makes the flow of control harder to follow, and has ill effects on testability. Platform dependencies should be handled using platform specific shadow files, preferably located in a general platform building-block.

exceptions:

- Conditional compilation of header by means of include guards: [ORG#001].
- Conditional application of platform dependent pragmas: see [POR#018].

Rule POR#022

Synopsis: Make sure all conversions from a value of one type to another of a narrower type do not slice off significant data
Language: C++
Level: 2
Category: Portability

Rule POR#025

Synopsis: Floating point values shall not be compared using the == or != operators
Language: C++
Level: 2
Category: Portability

Description

Due to rounding errors, most floating-point numbers end up being slightly imprecise. As long as this imprecision stays small, it can usually be ignored. However, it also means that numbers expected to be equal (e.g. when calculating the same result through different correct methods) often differ slightly, and a simple equality test fails. For example:

```
float a = 0.15 + 0.15
float b = 0.1 + 0.2
if (a == b) // can be false!
```

Solutions to this problem that are **not** correct include:

```
!(a < b) && !(a > b)
```

and

```
fabs(a - b) < Double.Epsilon
```

Please read [Dawson] to understand how to solve this issue.

Rule POR#028

Synopsis: Always return a value from main
Language: C++
Level: 9
Category: Portability

Description

Note that the most portable values are EXIT_SUCCESS and EXIT_FAILURE, declared in <cstdlib>.

Rule POR#029

Synopsis: Do not depend on the order of evaluation of arguments to a function
Language: C++
Level: 1
Category: Portability

Rule POR#030

Synopsis: Both operands of the remainder operator shall be positive
Language: C++
Level: 5
Category: Portability

Description

Please note that a % b is implementation-defined for negative a or b, and undefined for b == 0. Also note that automatic verification of this rule is often impractical in case of signed types.

Rule POR#031

Synopsis: Do not depend on implementation defined behavior of shift operators for built-in types
Language: C++
Level: 5
Category: Portability

Description

The right operand of a shift operator shall not be negative. The left operand of a shift operator shall not be signed. Left shift operators may cause information loss by truncation.

Rule POR#032

Synopsis: Use nullptr instead of 0 or NULL for a null pointer
Language: C++
Level: 9
Category: Portability

Description

Wrong example:

```
ptr = NULL;
```

Correct example:

```
ptr = nullptr;
```

Rule POR#033

Synopsis: Do not make assumptions on the size of int
Language: C++
Level: 2
Category: Portability

Description

This depends on the compiler and platform used.

Rule POR#037

Synopsis: Avoid the use of #pragma warning directive.
Language: C++
Level: 1
Category: Portability

Description

To keep the code warning free, using #pragma warning directive is not allowed.

Exception: #pragma warning is allowed in header files *stdafx.h, such that external files can be included, precompiled and controlled.

Rule POR#038

Synopsis: Make sure that the sizes and types of functions are well-defined if exported to other languages
Language: C++
Level: 4
Category: Portability

Description

When exporting functions for use from other languages, make sure that the sizes of arguments to these functions and the type of the return value are well-defined. Use the types of known size that are provided by your language implementation (e.g. uint32_t in C/C++11) or make use of local typedefs that are available in your environment. Do not use types such as int, short or long directly because these might have another size for other languages. For instance a long is 4 bytes in C++ on most platforms and 8 bytes in C#.

Style

This chapter concerns the style or layout of the code. A consistent style increases the readability of the code.

Background information: Operators are operations that are performed, operands are the arguments of these operations. E.g. in "a++", the operator is "++" and the operand is "a". A Unary operator is an operator that has one argument, a binary operator is an operator that takes two arguments. E.g. "<=" and "&&" are binary operators, whereas "!" is a unary operator. A ternary operator is an operator that takes three arguments,e.g. the conditional operator "?".

Rules

STY#002	Always use parentheses to clarify the order of expression evaluation
STY#017	If a method/function has no formal parameter, do not use the keyword void
STY#020	Use std::function instead of function pointers
STY#024	The name of an #include guard shall contain at least the name of the header file.
STY#025	Do not use letters that can be mistaken for digits, and vice versa
STY#029	Always provide an access specifier for class members

Rule STY#002

Synopsis: Always use parentheses to clarify the order of expression evaluation
Language: C++
Level: 9
Category: Style

Description

Do not base the proper working on the C++ implicit evaluation order. Parentheses in expressions improve the readability.

Rule STY#017

Synopsis: If a method/function has no formal parameter, do not use the keyword void
Language: C++
Level: 9
Category: Style

Description

In C a function prototype without any parameters means that the function parameters are not specified. The void keyword is used in C to indicate that the function has no parameters. However, in C++ a method or function prototype without any parameters means that the method or function does not have any parameters. The void keyword is no longer needed.

Rule STY#020

Synopsis: Use std::function instead of function pointers
Language: C++
Level: 10
Category: Style

Description

A reason to use std::function is that it offers more flexibility to the user (caller of the function): std::function accepts member- and free functions and accepts functions with bound extra arguments.

Wrong example:

```
int foo(int x) { return x; }
int (*foo_func)(int) = &foo;
```

Right example:

```
int foo(int x) { return x; }
std::function<int(int)> foo_func = &foo;
```

more elegant:

```
std::function<int(int)> foo_func = [](int x) { return x; };
```

even more elegant:

```
auto foo_func = [](int x) { return x; };
```

Rule STY#024

Synopsis: The name of an #include guard shall contain at least the name of the header file.
Language: C++
Level: 10
Category: Style

Description

If projects have files with the same name in different directories, it is not possible to include both files. The guard of an internal header file must have an additional underscore appended at the end. This is to prevent duplicate guard names when a certain interface is specified by both an internal and external header file that have the same name. See also [ORG#001].

Rule STY#025

Synopsis: Do not use letters that can be mistaken for digits, and vice versa
Language: C++
Level: 9
Category: Style

Rule STY#029

Synopsis: Always provide an access specifier for class members
Language: C++
Level: 9
Category: Style

Literature

Abrahams
Title: Exception-Safety in Generic Components
Author: David Abrahams
http://www.boost.org/more/generic_exception_safety.html

BigBallOfMud
Title: Big Ball of Mud
Author: Brian Foote & Joseph Yoder
Year: 1999
http://www.laputan.org/mud/

C++ Core Guidelines
Title: C++ Core Guidelines
Author: Bjarne Stroustrup and Herb Sutter
http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html

Dalvander
Title: Condition Variable Spurious Wakes
Author: Anders Dalvander
Year: 2008
https://www.justsoftwaresolutions.co.uk/threading/condition-variable-spurious-wakes.html

Dawson
Title: Comparing Floating Point Numbers, 2012 Edition
Author: Bruce Dawson
Year: 2012
http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

Ellemtel
Title: Programming in C++, Rules and Recommendations
Author: Mats Henricson & Erik Nyquist
Year: 1990
ISBN: M 90 0118 Uen

Gamma
Title: Design Patters, Elements of Reusable Object-Oriented Software
Author: Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides
Year: 1995
Publisher: Addison-Wesley
ISBN: ISBN 0-201-63361-2

Hatton
Title: Philips Medical Systems C Coding Standard
Author: Les Hatton
Year: 1998
ISBN: XK080-95075

ISC++
Title: Industrial Strength C++
Author: Mats Henricson & Erik Nyquist
Year: 1997
Publisher: Prentice Hall PTR

ISBN: ISBN 0-13-120965-5

Knatten
Title: Always catch exceptions by reference
Author: Anders Schau Knatten
Year: 2010
https://blog.knatten.org/2010/04/02/always-catch-exceptions-by-reference/

Liskov
Title: Data Abstraction and Hierarchy
Author: Barbara Liskov
Year: 1988
Publisher: SIGPLAN Notices, 23, 5

Meyers
Title: Effective C++, 2nd Edition
Author: Scott Meyers
Year: 1997
Publisher: Addison-Wesley
ISBN: ISBN 0-201-92488-9

Schaick
Title: CIS C++ Development - Coding Standard
Author: Edwin van Schaick
Year: 2004
ISBN: XLR-060-02-1580

StackOverflow
Title: StackOverflow
http://www.stackoverflow.com

Stroustrup
Title: The C++ Programming Language (Third Edition)
Author: Bjarne Stroustrup
Year: 1997
Publisher: Addison-Wesley
ISBN: ISBN 0-201-88954-4
http://www.research.att.com/~bs/3rd.html

StroustrupDE
Title: The Design and Evolution of C++
Author: Bjarne Stroustrup
Year: 1994
Publisher: Addison-Wesley
ISBN: ISBN 0-201-54330-3
http://www.research.att.com/~bs/dne.html

Tongeren
Title: PMS-MR C++ Coding Standard
Author: Bart van Tongeren
Year: 2002
ISBN: XJS-154-1215/3.1.0

Welch

Title: An Exception-Based Assertion Mechanism for C++
Author: David Welch & Scott Strong
Year: JOOP July/Aug 1998