The Motor Industry Software Reliability Association

MISRA C++:2008

Guidelines for the use of the C++ language in critical systems

June 2008

First published June 2008

MIRA Limited
Watling Street
Nuneaton
Warwickshire CV10 0TU
UK

www.misra-cpp.com

ISBN 978-1-906400-03-3 paperback
ISBN 978-1-906400-04-0 PDF

Printed by Hobbs the Printers Ltd

British Library Cataloguing in Publication Data.

A catalogue record for this book is available from the British Library

The Motor Industry Software Reliability Association

MISRA C++:2008

Guidelines for the use of the C++ language in critical systems

June 2008

MISRA Mission Statement: To provide assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software.

MISRA, The Motor Industry Software Reliability Association, is a collaboration between vehicle manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety-related electronic systems in road vehicles and other embedded systems. To this end MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

www.misra.org.uk

Disclaimer

Adherence to the requirements of this document does not in itself ensure error-free robust software or guarantee portability and re-use.

Compliance with the requirements of this document, or any other standard, does not of itself confer immunity from legal obligations.

Foreword

"*C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.*" — Bjarne Stroustrup

Few could have predicted the effect that MISRA C would have within embedded systems engineering. Since its launch in 1998, it has become the dominant coding standard used for the development of critical systems with the C programming language.

Given this success, the fact that C++ is now being used within critical systems (e.g. the Joint Strike Fighter, jet-engine controllers and medical systems), and that there is currently no universally accepted set of guidelines for its use in these systems, MISRA committed itself to the development of a similar set of guidelines for C++. To that end, the MISRA C++ Working Group was established towards the end of September 2005. Its objectives were to:

- Produce, using techniques similar to those within MISRA C, a C++ subset suitable for use in critical systems;
- Gather existing C++ guidelines from many diverse sources into a single repository;
- Add new guidance so as to significantly enhance the state-of-the-art;
- Establish a single, generic set of guidelines for the use of C++ in critical systems;
- Produce guidelines that are understandable to the majority of programmers.

The work to produce the guidelines made a rapid start, and was greatly assisted by the many in-house coding standards that were made available to the group — thanks are due to all those who contributed. These, and the guidelines available from other sources, formed a solid foundation on which to base many rules.

Focus then moved on to the production of guidelines for Templates, Inheritance and Exception Handling, with these areas being specifically targeted as the existing state-of-the-art did not provide adequate coverage. The issues associated with Unnecessary Constructs were also selected for investigation.

This document contains the results of these activities. The group hopes that MISRA C++ will go on to become as successful and widely-adopted as MISRA C.

Finally, I would like to give my personal thanks to all of those who sat on the Working Group. I have, as always, learnt a lot from them during the development process. I just hope I have managed to put enough back into the project to repay part of the debt I owe them all.

Chris Tapp, BSc (Hons), MIEE
MISRA C++ Chairman

15 April 2008

Acknowledgements

The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document:

Richard Corden	Programming Research Ltd
Mike Hennell	LDRA Ltd
Derek Jones	Knowledge Software Ltd
Keith Longmore	Lotus Cars Ltd
Clive Pygott	QinetiQ Ltd
Chris Tapp	Keylevel Consultants Ltd

The MISRA consortium also wishes to acknowledge contributions from the following individuals during the development and review process:

Dave Banham	Al Grant	Frank Martinez	John Ridgway
Martin Beeby	Christopher Hall	Jason Masters	Iris Rödder
Fergus Bolger	Frank Haug	Jürgen Mottok	Walter Schilling
Martin Bonner	Stefan Heinzmann	Chris Mycock	Ben Smith
Michael R. Bossert	Robert Hooper	Tadanori Nakagawa	Andreas Stangl
Antonio Cavallo	Elmar Hufschmid	Hans Odeberg	Toshihiro Tajima
Ian Chalinder	Paul Jeary	Charles Osborn	David Ward
Kwok Chan	Josef Kollar	Rob Pearce	Andrew Warren
Valéry Creux	Albert Kreitmeyr	PremAnand M Rao	Andrew Watson
David Crocker	Fred Long	Derek Reinhardt	Ashley Wise
Thomas M. Galla	Andreas Ludwig	David Reversat	

Contents

Contents (continued)

Contents (continued)

Contents (continued)

1. Background

1. Background

1.1 The use of C++ in critical systems

The C++ programming language [1] is growing in importance and use for critical systems. This is due largely to the inherent language flexibility, the extent of support and its potential for portability across a wide range of hardware. Specific reasons for its use include:

- C++ gives good support for the high-speed, low-level, input/output operations, which are essential to many embedded systems.
- The increased complexity of applications makes the use of a high-level language more appropriate than assembly language.
- C++ compilers can generate code with similar size and RAM requirements to those of C.
- C++ enables object-oriented design methods to be used.
- A growth in portability requirements caused by competitive pressures to reduce hardware costs by porting software to new, and/or lower cost, processors at any stage in a project lifecycle.
- A growth in the use of automatically-generated C++ code from modelling packages.
- Increasing interest in open systems and hosted environments for which C++ is a possible language selection.

1.2 Language insecurities and the C++ language

No programming language can guarantee that the final executable code will behave exactly as the programmer intended. There are a number of problems that can arise with any language, and these are broadly categorized below. Examples are given to illustrate insecurities in the C++ language.

1.2.1 The developer makes mistakes

Developers make errors, which can be as simple as mis-typing a variable name, or might involve something more complicated such as misunderstanding an algorithm. The programming language has a bearing on this type of error. Firstly, the style and expressiveness of the language can assist or hinder the programmer in thinking clearly about the algorithm. Secondly, the language can make it easy or hard for typing mistakes to turn one valid construct into another valid (but unintended) construct. Thirdly, the compiler may or may not detect errors when they are made.

Firstly, in terms of style and expressiveness C++ can be used to write well laid out, structured and expressive code. It can also be used to write perverse and extremely hard-to-understand code. Clearly the latter is not acceptable in a safety-related system.

Secondly, the syntax of C++ is such that it is relatively easy to make typing mistakes that lead to perfectly valid code. For example, it is all too easy to type “=” (assignment) instead of “==” (logical comparison) and the result is nearly always valid (but wrong), while an extra semi-colon on the end of an *if* statement can completely change the logic of the code.

Thirdly, the philosophy of C++ is to assume that the developers know what they are doing, which can mean that if errors are made they are allowed to pass unnoticed by the compiler. An area in which C++ is particularly weak (though better than C) in this respect is that of “type checking”. C++ will not object, for example, if the programmer tries to store a floating-point number in an

object of type bool. Most such mismatches are simply forced to become compatible. If C++ is presented with a square peg and a round hole it does not complain, but makes them fit!

1.2.2 The developer misunderstands the language

Developers can misunderstand the effect of constructs in a language. Some languages are more open to such misunderstandings than others.

There are a number of areas of the C++ language that are prone to developer-introduced errors. For example, the rules for operator precedence are well defined but complex, and it is easy for a developer to make incorrect assumptions in an expression.

1.2.3 The compiler does not do what the developer expects

If a language has features that are not completely defined, or are ambiguous, then a developer can assume one thing about the meaning of a construct, while the compiler may interpret it quite differently.

There are many areas of the C++ language that are not completely defined, and so behaviour may vary from one compiler to another. In some cases the behaviour can vary even within a single compiler, depending on the context. ISO/IEC 14882:2003 [1] contains numerous issues that may vary in this way. However, it does not list them in the way that the C standard does in its "Portability issues" annex. To aid in ensuring coverage of the issues and to allow traceability, they have been extracted and are shown in Appendix B.

1.2.4 The compiler contains errors

A language compiler (and associated linker etc.) is itself a software tool. Compilers may not always compile code correctly. They may, for example, not comply with the language standard in certain situations, or they may simply contain "bugs".

Because there are aspects of the C++ language that are hard to understand, compiler writers have been known to misinterpret the standard and implement it incorrectly. Some areas of the language are more prone to this than others. In addition, compiler writers sometimes consciously choose to vary from the standard.

1.2.5 Run-time errors

A somewhat different language issue arises with code that has compiled correctly, but due to the particular data supplied to it, causes errors during the execution of the code. Languages can build run-time checks into the executable code to detect many such errors and take appropriate action.

C++ is generally poor in providing run-time checking. This is one of the reasons why the code generated by C++ tends to be small and efficient, but there is a penalty to pay in terms of detecting errors during execution. C++ compilers generally do not provide run-time checking for such common problems as arithmetic exceptions (e.g. divide by zero), overflow, validity of addresses for pointers, or array bound errors.

1.3 The use of C++ for safety-related systems

It should be clear from Section 1.2 that great care needs to be exercised when using C++ within safety-related systems. Because of the kinds of issues identified above, various concerns have

been expressed about the use of C++ in safety-related systems. Certainly, it is clear that the full C++ language should not be used for programming safety-related systems.

However, in its favour, C++ is mature and consequently well-analysed and tried in practice. Therefore some of its deficiencies are known and understood. Additionally, there is a large amount of tool support available commercially which can be used to statically check the C++ source code and warn the developer of the presence of many of the problematic aspects of the language.

If, for practical reasons, it is necessary to use C++ on a safety-related system, then the use of the language must be constrained to avoid, as far as is practicable, those aspects of the language which do give rise to concerns. This document provides one such set of constraints (often referred to as a "language subset").

Note that assembly language is no more suitable for safety-related systems than C++ and, in some respects, is worse. Use of assembly language in safety-related systems is not recommended, and generally if it is to be used then it needs to be subject to stringent constraints.

1.4 C++ standardization

The standard used for this document is the C++ programming language as defined by ISO/IEC 14882:2003 [1].

2. The Vision

2. The vision

2.1 Rationale for the production of MISRA C++

The MISRA consortium published its *Development Guidelines for Vehicle Based Software* [2] in 1994, which describes the full set of measures that should be used in safety-related embedded software development for vehicle systems. In particular, the choices of language, compiler and language features to be used, in relationship with safety integrity level (SIL), are recognized to be of major importance. Section 3.2.4.3 (b) and Table 3 of the MISRA Guidelines [2] address this. One of the measures recommended is the use of a subset of a standardized language, which is already established practice in the automotive, aerospace, nuclear and defence industries. Similarly, other safety-related systems standards require subsets of programming languages in general (e.g. [12] Part 3 Table A.3) even if they do not explicitly require a subset of C++. This document therefore addresses the definition of a suitable subset of C++.

2.2 Objectives of MISRA C++

In publishing this document regarding the use of the C++ programming language, the MISRA consortium is not intending to promote the use of C++. Rather, it recognizes the already widespread use of C++, and this document seeks only to promote the safest possible use of the language.

It is the hope of the MISRA consortium that this document will gain industry acceptance and that the adoption of a safer subset will become established as best practice by vehicle manufacturers, component suppliers and other industrial sectors. It should also encourage training and enhance competence in general C++ programming and in this specific subset, at both an individual level and a company level.

Great emphasis is placed on the use of static checking tools to enforce compliance with the subset and it is hoped that this too will become common practice by the developers of critical systems.

Although much has been written about the advantages and disadvantages of various programming languages, this information is not well-known among developers. This document makes such information readily available, which should lead to an increase in the awareness of language-choice issues among engineers and managers.

3. Scope

3. Scope

3.1 Base language issues

The MISRA Guidelines [2] (Table 3) and IEC 61508 [12] require that “a restricted subset of a standardized structured language” be used for critical systems. This means that the language must only be used as defined in ISO/IEC 14882:2003 [1], precluding the use of language extensions.

3.2 Issues not addressed

Issues of style and code metrics are somewhat subjective. It would be hard for any group of people to agree on what was appropriate, and it would be inappropriate for MISRA to give definitive advice. It is, however, important that suitable style guidelines and appropriate metrics and limits are selected.

The MISRA consortium is not in a position to recommend particular vendors or tools to enforce the restrictions adopted. The user of this document is free to choose tools, and vendors are encouraged to provide tools to enforce the rules. The onus is on the user of this document to demonstrate that their chosen tool set(s) enforces the rules adequately.

3.3 Applicability

This document is designed to be applied to production code in critical systems.

In terms of the implementation defined by ISO/IEC 14882:2003 [1] §1.4(7), this document is aimed at a *freestanding implementation*, although it also addresses library issues since standard libraries will often be supplied with an embedded compiler. The requirements of this document will not necessarily be applicable in their entirety to a *hosted implementation*, although many of the requirements will help create higher quality software in that context.

3.4 Prerequisite knowledge

This document is not intended to be an introduction or training aid to the subjects it embraces. It is assumed that readers of this document are familiar with the ISO C++ programming language standard and associated tools, and also have access to the primary reference documents. It also assumes that developers have received appropriate training and are competent C++ language programmers.

3.5 Library issues

In general, it is preferable that library code should fully satisfy the full set of MISRA C++ Rules.

However, it is recognized that there are situations when this ideal cannot be achieved. In these cases it may be possible to demonstrate that there is only a small safety risk incurred if deviations are raised against these rules (e.g. Rule 0–1–5, Rule 0–1–10, Rule 14–7–1 and Rule 14–7–2). In these cases, the deviation raised shall justify that alternative strategies are in place (e.g. by identifying which library functions are intended to be used and then showing that these, and only these, have been used).

3.6 Auto-generated code issues

It is recognized that any C++ code produced by an automatic code generation tool introduces another level of complexity into the software development process. In general, any C++ code that is produced by such a tool (either automatically by the tool or that is included by the tool but which is provided by a developer) should fully satisfy the full set of MISRA C++ rules.

However, an automatic code generation tool could potentially detect MISRA C++ violations (e.g. a missing *default* clause in a *switch* statement) and supply code to ensure that the violation is eliminated. This is considered to be undesirable as the responsibility for ensuring compliance against MISRA C++ should lie with the developer (for example, could a tool always provide an appropriate default action?).

MISRA AC INT [3] contains an introduction to the MISRA guidelines for the use of model-based development and automatic code generation. Note that, at the time of publication of this document, MISRA AC does not include specific guidance on the use of C++ as a target language.

Additionally, if a code-generating tool is to be used, then it will be necessary to select an appropriate tool and undertake validation. Apart from suggesting that adherence to the requirements of this document may provide one criterion for assessing a tool, no further guidance is given on this matter and the reader is referred to the HSE recommendations for COTS [4].

Automatically-generated code must be treated in the same manner as manually produced code for the purpose of validation (See MISRA Guidelines [2] §3.1.3, Planning for V&V).

4. Using MISRA C++

4.1 The software engineering context

Using a programming language to produce source code is only one activity in the software development process. Adhering to best practice in this one activity is of very limited value if the other commonly accepted development issues are not addressed. This is especially true for the production of safety-related systems. These issues are all addressed in the MISRA Guidelines [2] and, for example, include:

- Documented development process;
- Quality system capable of meeting the requirements of ISO 9001/ISO 90003/TickIT [5], [6], [7];
- Project management;
- Configuration management;
- Hazard analysis;
- Requirements;
- Design;
- Coding;
- Verification;
- Validation.

It is necessary for the software developers to justify that the whole of their development process is appropriate for the type of system they are developing. This justification will be incomplete unless a hazard analysis activity has been performed to determine the SIL allocated to the system.

4.2 The programming language and coding context

Within the coding phase of the software development process, the language subset is only one aspect of many and again adhering to best practice in this aspect is of very limited value if the other issues are not addressed. Key issues, following choice of language, are:

- Training;
- Style guide;
- Compiler selection and validation;
- Checking tool validation;
- Metrics;
- Test coverage.

All decisions made on these issues, including the reasons for those decisions, need to be documented, and appropriate records should be kept for any activities performed. Such documentation may then be included in a safety justification, if required.

4.2.1 Training

In order to ensure an appropriate level of skill and competence on the part of those who produce the C++ source code, formal training should be provided for:

- The use of the C++ programming language for embedded applications;
- The use of the C++ programming language for high-integrity and safety-related systems;
- The use of static checking tools used to enforce adherence to the subset.

4.2.2 Style guide

In addition to adopting the subset, an organization should also have an in-house style guide. This will contain guidance on issues that do not directly affect the correctness of the code but rather define a "house style" for the appearance of the source code. These issues are subjective and typically include:

- Code layout and use of indenting;
- Layout of braces "`{ }`" and block structures;
- Statement complexity;
- Naming conventions;
- Use of comments;
- Inclusion of company name, copyright notice and other standard file header information.

While some of the content of the style guide may only be advisory, some may be mandatory. However the enforcement of the style guide is outside the scope of this document.

For further information on style guides see [8].

4.2.3 Tool selection and validation

When choosing a compiler (which should be understood to include the linker), an ISO C++ compliant compiler should be used whenever possible. Where the use of the language is reliant on an *implementation-defined* feature (as identified in Appendix B) then the developer must benchmark the compiler to establish that the implementation is as documented by the compiler writer.

When choosing a static checking tool it is clearly desirable that the tool enforces as many of the rules in this document as possible. To this end it is essential that the tool is capable of performing checks across the whole program, and not only within a single source file. In addition, where a checking tool has capabilities to perform checks beyond those required by this document, it is recommended that the extra checks are used.

The compiler and the static checking tool are generally seen as "trusted" processes. This means that there is a certain level of reliance on the output of the tools. The developer must therefore ensure that this trust is not misplaced. Ideally this should be achieved by the tool supplier running appropriate validation tests. Note that, while it is possible to use a validation suite to test a compiler for an embedded target, no formal validation scheme exists at the time of publication of this document. In addition, the tools should have been developed to a quality system capable of meeting the requirements of ISO 9001/ISO 90003 [5], [6], [7].

It should be possible for the tool supplier to show records of verification and validation activities together with change records that show a controlled development of the software. The tool supplier should have a mechanism for:

- Recording faults reported by the users;
- Notifying existing users of known faults;
- Correcting faults in future releases.

The size of the existing user base together with an inspection of the faults reported over the previous 6 to 12 months will give an indication of the stability of the tool.

It is often not possible to obtain this level of assurance from tool suppliers, and in these cases the onus is on the developer to ensure that the tools are of adequate quality.

Some possible approaches the developer could adopt to gain confidence in the tools are:

- Perform some form of documented validation testing;
- Assess the software development process of the tool supplier;
- Review the performance of the tool to date.

The validation test could be performed by creating code examples to exercise the tools. For compilers this could consist of known good code from a previous application. For a static checking tool, a set of code files should be written, each breaking one rule in the subset and together covering as many as possible of the rules. For each test file the static checking tool should then find the non-conformant code. Although such tests would necessarily be limited, they would establish a basic level of tool performance.

It should be noted that validation testing of the compiler must be performed for the same set of compiler options, linker options and source library versions used when compiling the product code.

The use of additional static analysis checks, where available, is also recommended.

4.2.4 Source complexity metrics

The use of source code complexity metrics is highly recommended. These can be used to prevent unwieldy and un-testable code being written by looking for values outside of established norms. The use of tools to collect the data is also highly recommended. Many of the static checking tools that may be used to enforce the subset also have the capability for producing metrics data.

For details of possible source code metrics see "*Software Metrics: A Rigorous and Practical Approach*" by Fenton and Pfleeger [9] and the MISRA report on Software Metrics [10].

4.2.5 Test coverage

The expected statement coverage of the software should be defined before the software is designed and written. Code should be designed and written in a manner that supports high statement coverage during testing. The term "Design For Test" (DFT) has been applied to this concept in mechanical, electrical and electronic engineering. This issue needs to be considered during the activity of writing the code, since the ability to achieve high statement coverage is an emergent property of the source code.

Use of a subset, which reduces the number of implementation-dependent features and increases the rigour of module interface compatibility, can lead to software that can be integrated and tested with greater ease.

Balancing the following metrics can facilitate achieving high statement coverage:

- Code size;
- Cyclomatic complexity.

With a planned approach, the extra effort expended on software design, language use and design for test is more than offset by the reduction in the time required to achieve high statement coverage during test. See [10], [11].

4.3 Adopting the subset

In order to develop code that adheres to the subset the following steps need to be taken:

- Produce a compliance matrix which states how each rule is enforced;
- Produce a deviation procedure;
- Formalize the working practices within the quality management system.

4.3.1 Compliance matrix

In order to ensure that the source code written does conform to the subset it is necessary to have measures in place that check that none of the rules have been broken. The most effective means of achieving this is to use one or more of the static checking tools that are available commercially. Where a rule cannot be checked by a tool, then a manual review will be required.

In order to ensure that all the rules have been covered then a compliance matrix should be produced which lists each rule and indicates how it is to be checked. See Table 1 for an example, and see Appendix A for a summary list of the rules, which could be used to assist in generating a full compliance matrix.

Rule No.	Compiler 1	Compiler 2	Checking Tool 1	Checking Tool 2	Manual Review
Rule 0–1–1	warning 347				
Rule 0–1–2		error 25			
Rule 0–1–3			message 38		
Rule 0–1–4				warning 97	
Rule 0–1–5					Proc x.y

Table 1: Example compliance matrix

If the developer has additional local restrictions, these too can be added to the compliance matrix. Where specific restrictions are omitted, full justifications shall be given. These justifications must be fully supported by a C++ language expert together with manager level concurrence.

4.3.2 Deviation procedure

It is recognized that in some instances it may be necessary to deviate from the rules given in this document. For example, source code written to interface with the microprocessor hardware will inevitably require the use of proprietary extensions to the language.

In order for the rules to have authority, it is necessary that a formal procedure be used to authorize these deviations rather than an individual programmer having discretion to deviate at will. It is expected that the procedure will be based around obtaining a sign-off for each deviation, or class of deviation. The use of a deviation must be justified on the basis of both necessity and safety. While this document does not give, nor intend to imply, any grading of importance of each of the rules, it is accepted that some provisions are more critical than others. This should be reflected in the deviation procedure, where for more serious deviations greater technical competence is required to assess the risk incurred and higher levels of management are required to accept this increased risk. Where a formal quality management system exists, the deviation procedure should be a part of this system.

Deviations may occur for a specific instance, i.e. a one-off occurrence in a single file, or for a class of circumstances, i.e. a systematic use of a particular construct in a particular circumstance, for example the use of a particular language extension to implement an input/output operation in files that handle serial communications.

Strict adherence to all rules is unlikely and, in practice, deviations associated with individual situations, are admissible. There are two categories of deviation.

- Project Deviation: A Project Deviation is defined as a permitted relaxation of rule requirements to be applied in specified circumstances. In practice, Project Deviations will usually be agreed at the start of a project.
- Specific Deviation: A Specific Deviation will be defined for a specific instance of a rule violation in a single file and will typically be raised in response to circumstances that arise during the development process.

Project deviations should be reviewed regularly and this review should be a part of the formal deviation process.

Many, if not most, of the circumstances where rules need to be broken are concerned with input/output operations. It is recommended that the software be designed such that input/output concerns are separated from the other parts of the software. As far as possible Project Deviations should then be restricted to this input/output section of the code. Code subject to Project Deviations should be clearly marked as such.

The purpose of this document is to avoid problems by thinking carefully about the issues and taking all responsible measures to avoid the problems. The deviation procedure should not be used to undermine this intention. In following the rules in this document the developer is taking advantage of the effort expended by MISRA in understanding these issues. If the rules are to be deviated from, then the developer is obliged to understand the issues for themselves. All deviations, standing and specific, should be documented.

For example, if it is known beforehand that it will be difficult to adhere to a rule, the software developer should submit a written Project Deviation Request and agreement with the customer should be obtained prior to programming.

A Project Deviation Request should include the following:

- Details of the deviation, i.e. the rule that is being violated;
- Circumstances in which the need for the deviation arises;
- Potential consequences which may result from the deviation;
- Justification for the deviation;
- A demonstration of how safety is assured.

When the need for a deviation arises during or at the end of the development process, the software developer should submit a written Specific Deviation Request.

A Specific Deviation Request should include the following:

- Details of the deviation, i.e. the rule that is being violated;
- Potential consequences which may result from the deviation;
- Justification for the deviation;
- A demonstration of how safety is assured.

Detailed implementation of these procedures is left to the discretion of the user.

4.3.3 Formalization within quality system

The use of the subset, the static checking tools and deviation procedure should be described by formal documents within the quality management system. They will then be subject to the internal and external audits associated with the quality system and this will help ensure their consistent use.

4.3.4 Introducing the subset

Where an organization has an established C++ coding environment it is recommended that the requirements of this document be introduced in a progressive manner. It may take 1 to 2 years to implement all aspects of this document.

Where a product contains legacy code written prior to the use of the subset, it may be impractical to rewrite it to bring it into conformance with the subset. In these circumstances the developer must decide upon a strategy for managing the introduction of the subset (for example: all new modules will be written to the subset and existing modules will be rewritten to the subset if they are subject to a change which involves more than 30% of the non-comment source lines).

4.4 Claiming compliance

Compliance can only be claimed for a product and not for an organization.

When claiming MISRA C++ compliance for a product, a developer is stating that evidence exists to show:

- A compliance matrix has been completed which shows how compliance has been enforced;
- All of the C++ code in the product is compliant with the rules of this document or subject to documented deviations;
- A list of all instances of rules not being followed is being maintained, and for each instance there is an appropriately signed-off deviation;
- The issues mentioned in Section 4.2 have been addressed.

4.5 Continuous improvement

Adherence to the requirements of this document should only be considered as a first step in a process of continuous improvement. Users should be aware of the other literature on the subject (see references) and actively seek to improve their development process by the use of metrics.

5. Introduction to the rules

This section explains the presentation of the language guidelines (the rules) given in this part of the document. It serves as an introduction to the main content of the document as presented in that section.

5.1 Rule classification

Every rule is classified as "Required", "Advisory" or "Document", as described below. Beyond this basic classification the document does not give, nor intend to imply, any grading of importance of each of the rules. All "Required" rules should be considered to be of equal importance, as should all "Advisory", and all "Document" rules. The omission of an item from this document does not imply that it is less important. Furthermore for projects following a safety standard that uses SIL or a similar measure of risk reduction requirements, all rules are intended to be applied regardless of the SIL claimed for the software being developed.

The meanings of "Required", "Advisory" and "Document" rules are as follows.

5.1.1 Required rules

These are mandatory requirements placed on the developer. C++ code that is claimed to conform to MISRA C++ shall comply with every "Required" rule. Formal deviations must be raised where this is not the case.

5.1.2 Advisory rules

These are requirements placed on the developer that should normally be followed. However they do not have the mandatory status of "Required" rules. Note that the status of "Advisory" does not mean that these items can be ignored, but that they should be followed as far as is reasonably practical. Formal deviations are not necessary for "Advisory" rules, but may be raised if it is considered appropriate.

5.1.3 Document rules

These are mandatory requirements placed on the developer whenever the associated feature is used within code. Deviations are not permitted against this class of rule.

5.2 Organization of rules

The rules are organized under the section numbers of ISO/IEC 14882:2003 [1]. However there is inevitably overlap, with one rule possibly being relevant to a number of topics. Where this is the case, the rule has been placed under the first relevant topic.

5.3 Exceptions to the rules

Some rules contain an "Exception" section that lists one or more exceptional conditions under which the rule need not be followed. These exceptions effectively modify the headline rule.

5.4 Redundancy in the rules

There are a few cases within this document where a rule is given that refers to a language feature, which is banned or advised against elsewhere in the document. This is intentional. It may be that the developer chooses to use that feature, either by raising a deviation against a "Required" rule, or by choosing not to follow an "Advisory" rule. In this case, the second rule, constraining the use of that feature, becomes relevant.

5.5 Presentation of rules

The individual Rules are presented in the following format:

Rule <*number*> (<*category*>) <*headline text*>

[<*issue reference*>]

<normative text>

Where the fields are as follows:

- <*number*> Every rule has a unique number, consisting of three parts, **aa–bb–cc**.
  aa–bb gives the section number within ISO/IEC 14882:2003 [1] to which the rule relates. Note, because guidance is not given for every section in ISO/IEC 14882:2003 [1], the numbering of the designators **aa–bb** and hence of the section headings in this document are not contiguous.

  cc is a sequence number for rules related to the above section.

  Note that section **0–bb** contains general guidance that does not relate to any specific area within the standard.

  Sections of the form **aa–0** give guidance that relates to section **aa** of the standard, but which is not directly attributable to any particular language construct within that section.
- <*category*> is one of "Required", "Advisory" or "Document", as explained above.
- <*headline text*> is the rule itself.
- <*issue reference*> indicates the section and paragraph number within ISO/IEC 14882:2003 [1] of a specific language issue which is targeted by the rule. An explanation of these issues is given below.

Normative text is provided for each rule. This text gives, where appropriate, some explanation of the underlying issues being addressed by the rule, and examples of how to apply the rule.

The normative text is not intended as a tutorial in the relevant language feature, as the reader is assumed to have a working knowledge of the language. Further information on the language features can be obtained by consulting the relevant section of the language standard or other C++ language reference books. Where an Issue Reference is given, then the original issue raised in ISO/IEC 14882:2003 [1] may provide additional help in understanding the rule.

Within the rules and their supporting text the following font styles are used to represent C++ keywords and C++ code:

C++ *keywords* appear in italic text

C++ `code` appears in a mono-spaced font, either within other text or as

```
separate code fragments;
```

Note that where code is quoted, the fragments may be incomplete (for example an *if* statement without its body). This is for the sake of brevity.

In code fragments, in order to comply with Rule 3–9–2, the following *typedef*'d types have been assumed:

```
char_t      // plain 8 bit character
uint8_t     // unsigned 8 bit integer
uint16_t    // unsigned 16 bit integer
uint32_t    // unsigned 32 bit integer
int8_t      // signed 8 bit integer
int16_t     // signed 16 bit integer
int32_t     // signed 32 bit integer
float32_t   // 32 bit floating-point
float64_t   // 64 bit floating-point
```

Note that the `bool` and `wchar_t` types do not have *typedefs*.

Non-specific variable names are constructed to give an indication of the type. For example:

```
uint8_t     u8a;
int32_t     s32a;
```

5.6 Understanding the issue references

Where a rule is aimed at a specific issue within ISO/IEC 14882:2003 [1], then a reference to the standard is indicated in square brackets after the rule. This serves two purposes. Firstly, a reader may wish to gain a fuller understanding of the rationale behind the rule (for example when considering a request for a deviation) by consulting the standard. Secondly, this gives extra information about the nature of the problem.

Rules that do not have an Issue Reference may have originated from a contributing company's in-house standard, or have been suggested by a reviewer, or be widely accepted "good practice".

A key to the references, and advice on interpreting them, is given below.

Key to the source references

Reference	Source
Unspecified	*Unspecified behaviour* detailed within the C++ standard
Undefined	*Undefined behaviour* detailed within the C++ standard
Indeterminate	*Undefined behaviour* that arises due to an omission in the C++ standard of an explicit definition of behaviour
Implementation	*Implementation-defined behaviour* detailed within the C++ standard
NDR	Behaviour within the C++ standard for which *no diagnostic is required*
IEC 61508	IEC 61508:1998–2000 [12]

Where numbers follow the reference, they have the following meanings:

- For language issues, the section and paragraph number (if appropriate) of the relevant part of the standard are given, i.e. 1.2(3) would indicate Section 1.2, Paragraph 3.
- In other references, the relevant section number is given (unless stated otherwise).

Issue references

Where a rule is based on issues within ISO/IEC 14882:2003 [1], it is helpful for the reader to understand the distinction between "Unspecified", "Undefined", "Indeterminate", "Implementation-Defined" and "NDR" issues.

Unspecified

These are language constructs that must compile successfully, but in which the compiler writer has some freedom as to what the construct does. "Order of evaluation" is an example of this.

It is unwise to place any reliance on the compiler behaving in a particular way. The compiler need not even behave consistently across all uses of a construct.

Undefined

These are essentially programming errors for which the compiler is not obliged to issue a diagnostic (error message). Examples are invalid escape sequences or attempting to modify a string literal.

These are particularly important from a safety point of view, as they represent programming errors that may not necessarily be trapped by the compiler.

Indeterminate

This is similar to "Undefined", but where ISO/IEC 14882:2003 [1] omits an explicit definition of behaviour.

Implementation-defined

These are similar to the "Unspecified" issues, the main difference being that the compiler writer must take a consistent approach and document it. In other words, the functionality can vary from one compiler to another, making code non-portable, but on any one compiler the behaviour should be well defined. An example of this is the behaviour of the integer division and remainder operators "`/`" and "`%`" when applied to one positive and one negative integer.

These tend to be less critical from a safety point of view, provided the compiler writer has fully documented their approach and then implemented it consistently. It is advisable to avoid these issues where possible.

NDR

"No diagnostic required" (NDR) conditions are those that may lead to program errors, but for which the complier is not required to issue a diagnostic (error message).

5.7 Scope of rules

While the majority of rules can be applied within a single translation unit, all rules shall be applied with the widest possible interpretation.

In general, the intent is that all the rules shall be applied to templates. However, some rules are only meaningful for instantiated templates.

Unless otherwise specified, all rules shall apply to *implicitly-declared* or *implicitly-defined* special member functions (e.g. default constructor, copy constructor, copy assignment operator and destructor).

6. Rules

Appendix C contains a glossary of terms that are used in the formation of the rules that are presented within this section.

6.0 Language independent issues

6.0.1 Unnecessary constructs

An unnecessary construct by itself is benign and therefore leads to no faults, but it is a defect. However, it is possible that these defects may be due to errors and hence they may be coupled to, or indicate faults. The following rules address the need to reduce the number of these defects.

The attempt to remove all such constructs may reveal the underlying fault if one is present. The absence of such constructs leads to code that is more readable, faster executing and more easily maintained. In addition, software analysis tools can produce more accurate results. The presence of infeasible code, for example, may lead to false positive and false negative messages from static analysis tools. In dynamic analysis, it prevents the achievement of required coverage metrics. Readability is impaired, as also is maintainability.

Rule 0–1–1 (Required) A *project* shall not contain *unreachable code*.

Rationale

Code is unreachable if there is no syntactic (control flow) path to it. If such code exists, it is unclear if this is intentional or simply that an appropriate path has been accidentally omitted.

Compilers may choose not to generate code for these constructs, meaning that, even if the *unreachable code* is intentional, it may not be present in the final executable code.

Missing statements, often caused by editing activities, are a common source of *unreachable code*.

Example

```
int16_t with_unreach ( int16_t para )
{
   int16_t local;
   local = 0;
   switch ( para )
   {
      local = para;   // unreachable – Non-compliant
      case 1:
      {
         break;
      }
      default:
      {
         break;
      }
   }
   return para;
   para++;            // unreachable – Non-compliant
}
```

6. Rules

Rule 0–1–2	(Required)	A *project* shall not contain *infeasible paths*.

Rationale

Infeasible paths occur where there is a syntactic path but the semantics ensure that the control flow path cannot be executed by any input data. One of the major problems here is the explosion of *infeasible paths* caused by:

- *if … else* statement sequences;
- Sequences of poorly chosen loop constructs

Errors in conditions and poorly designed logic contribute to this problem. It is always possible to rewrite the code to eliminate these constructs. This process may then reveal faults.

There is the possibility that protective coding techniques generate infeasible code. This code is usually executable (and hence feasible) in a unit testing environment.

Example

```
void infeas ( uint8_t para, uint8_t outp )
{
   // The condition below will always be true hence the path
   // for the false condition is infeasible. Non-compliant.
   if ( para >= 0U )
   {
      outp = 1U;
   }
   // The following if statement combines with the if
   // statement above to give four paths. One from
   // the first condition is already infeasible and
   // the condition below combined with assignment above
   // makes the false branch infeasible. There is therefore
   // only one feasible path through this code.
   if ( outp == 1U )
   {
      outp = 0U;
   }
}
enum ec { RED, BLUE, GREEN } col;
if ( col <= GREEN )       // Non-compliant – always true
{
   // Will always get here
}
else
{
   // Will never get here
}
// The following ifs exhibit similar behaviour.
// Note that u16a is a 16-bit unsigned integer
// and s8a is an 8-bit signed integer.
if ( u16a < 0U )          // Non-compliant – u16a is always >= 0
if ( u16a <= 0xffffU )    // Non-compliant – always true
```

```
if ( s8a < 130 )                        // Non-compliant – always true
if ( ( s8a < 10 ) && ( s8a > 20 ) )     // Non-compliant – always false
if ( ( s8a < 10 ) || ( s8a > 5 ) )      // Non-compliant – always true
// Nested conditions can also cause problems
if ( s8a > 10 )
{
   if ( s8a > 5 )                 // Non-compliant, unless s8a volatile
   {
      // Will always get here.
   }
}
```

Rule 0–1–3	(Required)	A *project* shall not contain *unused* variables.

Rationale

Variables declared and never *used* in a *project* constitute noise and may indicate that the wrong variable name has been used somewhere. Removing these declarations reduces the possibility that they may later be used instead of the correct variable.

If padding is used within bit-fields, then the padding member should be unnamed to avoid violation of this rule.

Example

```
extern void usefn ( int16_t a, int16_t b );
class C
{
   ...
};
C c;                           // Non-compliant - unused
void withunusedvar ( void )
{
   int16_t unusedvar;          // Non-compliant – unused
   struct s_tag
   {
      signed int a   : 3;
      signed int pad : 1;      // Non-compliant – should be unnamed
      signed int b   : 2;
   } s_var;
   s_var.a = 0;
   s_var.b = 0;
   usefn ( s_var.a, s_var.b );
}
```

Rule 0–1–4	(Required)	A *project* shall not contain non-volatile *POD* variables having only one *use*.

Rationale

With the exception of volatile variables, variables declared and used only once do not contribute to program computations. A *use* is either an assignment (explicit initialization) or a reference.

These variables are essentially noise but their presence may indicate that the wrong variable has been used elsewhere. Missing statements contribute to this problem.

Example

```
const int16_t x = 19;         // Compliant
const int16_t y = 21;         // Non-compliant

void usedonlyonce ( void )
{
   int16_t once_1 = 42;       // Non-compliant
   int16_t once_2;

   once_2 = x ;               // Non-compliant
}
```

Note that x is compliant as there are two uses, firstly when initialized and secondly when assigned to once_2.

Rule 0–1–5	**(Required)**	**A *project* shall not contain *unused* type declarations.**

Rationale

If a type is declared but not *used*, then it is unclear to a reviewer if the type is redundant or it has been left unused by mistake.

See Section 3.5 for associated library issues.

Example

```
int16_t unusedtype()
{
   typedef int16_t local_Type;      // Non-compliant

   return 67;
}
```

Rule 0–1–6	**(Required)**	**A *project* shall not contain instances of non-volatile variables being given values that are never subsequently *used*.**

Rationale

Technically known as a *DU dataflow anomaly*, this is a process whereby a variable is given a value that is subsequently never *used*. At best this is inefficient, but may indicate a genuine problem. Often the presence of these constructs is due to the wrong choice of statement aggregates such as loops.

Exception

Loop control variables (see Section 6.6.5) are exempt from this rule.

Example

```
int16_t critical ( int16_t i, int16_t j )
{
   int16_t result = 0;
   int16_t k      = ( 3 * i ) + ( j * j );
```

```
   // Should k be checked here?
   if ( f2 ( ) )
   {
      // k will only be tested here if f2 returns true
      // Initialization of k could be moved here
      if ( k > 0 )
      {
         throw ( 42 );
      }
   }
   // Non-compliant – value of k not used if f2 ( ) returns false
   return ( result );
}
void unusedvalue ( int16_t arr[ 20 ] )
{
   int16_t j;
   j = 2;
   for ( int16_t i = 1; i < 10; i++ )
   {
      arr[ i ] = arr[ j ];
      j++;          // Non-compliant – the value assigned to j
   }                // on the final loop is never used.
}
void nounusedvalue ( int16_t arr[ 20 ] )
{
   for ( int16_t i = 1; i < 10; i++ )
   {
      arr[ i ] = arr[ i + 2 ];
   }
}
```

Rule 0–1–7	**(Required)**	**The value returned by a function having a non-*void* return type that is not an overloaded operator shall always be *used*.**

Rationale

In C++ it is possible to call a function without *using* the return value, which may be an error. The return value of a function shall always be *used*.

Overloaded operators are excluded, as they should behave in the same way as built-in operators.

Exception

The return value of a function may be discarded by use of a `(void)` cast.

Example

```
uint16_t func ( uint16_t para1 )
{
   return para1;
}
```

```
void discarded ( uint16_t para2 )
{
   func ( para2 );         // value discarded – Non-compliant
   (void)func ( para2 );   // Compliant
}
```

See also

Rule 5–2–4

Rule 0–1–8	(Required)	All functions with *void* return type shall have external side effect(s).

Rationale

A function which does not return a value and which does not have external side effects will only consume time and will not contribute to the generation of any outputs, which may not meet developer expectations.

The following are examples of external side effects:

- Reading or writing to a file, stream, etc.;
- Changing the value of a non local variable;
- Changing the value of an argument having reference type;
- Using a volatile object;
- Raising an exception.

Example

```
void pointless ( void )   // Non-compliant – no external side effects
{
   int16_t local;
   local = 0;
}
```

Rule 0–1–9	(Required)	There shall be no *dead code*.

Rationale

Any executed statement whose removal would not affect program output constitutes *dead code* (also know as *redundant code*). It is unclear to a reviewer if this is intentional or has occurred due to an error.

Example

```
int16_t has_dead_code ( int16_t para )
{
   int16_t local = 99;
   para  = para + local;
   local = para;         // dead code – Non-compliant
```

```
   if ( 0 == local )     // dead code – Non-compliant
   {                     // dead code – Non-compliant
      local++;           // dead code – Non-compliant
   }                     // dead code – Non-compliant
   return para;
}
```

Rule 0–1–10	(Required)	Every defined function shall be called at least once.

Rationale

Functions or procedures that are not called may be symptomatic of a serious problem, such as missing paths.

Note that an unused prototype is not a violation of this rule.

See Section 3.5 for associated library issues.

Example

```
void f1 ( )
{
}
void f2 ( )       // Non-compliant
{
}
void f3 ( );      // Compliant prototype
int32_t main ( )
{
   f1 ( );
   return ( 0 );
}
```

Rule 0–1–11	(Required)	There shall be no *unused* parameters (named or unnamed) in non-virtual functions.

Rationale

Unused function parameters are often due to design changes and can lead to mismatched parameter lists.

Exception

An unnamed parameter in the definition of a function that is used as a *callback* does not violate this rule.

Example

```
typedef int16_t ( * CallbackFn )( int16_t a, int16_t b );
int16_t Callback_1 ( int16_t a, int16_t b )   // Compliant
{
   return a + b;
}
```

```
int16_t Callback_2 ( int16_t a, int16_t b )   // Non-compliant
{
   return a;
}
int16_t Callback_3 ( int16_t, int16_t b )     // Compliant by exception
{
   return b;
}
void Dispatch ( int16_t n,
                int16_t a,
                int16_t b,
                int16_t c,  // Non-compliant
                int16_t )   // Non-compliant if Dispatch not a callback.
{
   CallbackFn pFn;
   switch ( n )
   {
      case  0: pFn = &Callback_1; break;
      case  1: pFn = &Callback_2; break;
      default: pFn = &Callback_3; break;
   }
   ( *pFn )( a, b );
}
```

See also

Rule 0–1–12

Rule 0–1–12 (Required)	There shall be no *unused* parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Rationale

Unused function parameters are often due to design changes and can lead to mismatched parameter lists.

Example

```
class A
{
public:
   virtual void withunusedpara    ( uint16_t * para1,
                                    int16_t    unusedpara ) = 0;
   virtual void withoutunusedpara ( uint16_t * para1,
                                    int16_t  & para2      ) = 0;
};
```

```
class B1: public A
{
public:
   virtual void withunusedpara ( uint16_t * para1,
                                 int16_t    unusedpara )
   {
      *para1 = 1U;
   }
   virtual void withoutunusedpara ( uint16_t * para1,
                                    int16_t  & para2 )
   {
      *para1 = 1U;
   }
};
class B2: public A
{
public:
   virtual void withunusedpara ( uint16_t * para1,
                                 int16_t    unusedpara )
   {
      *para1 = 1U;
   }
   virtual void withoutunusedpara ( uint16_t * para1,
                                    int16_t  & para2 )
   {
      para2 = 0;
   }
};
```

See also

Rule 0–1–11

6.0.2 Storage

Rule 0–2–1	**(Required)**	**An object shall not be assigned to an overlapping object.**

[Undefined 5.17(8)]

Rationale

Assigning between objects that have an overlap in their physical storage leads to *undefined behaviour*.

Example

```
struct s
{
   int16_t m1 [ 32 ];
};
struct t
{
   int32_t  m2;
   struct s m3;
};
void fn ( )
{
   union             // Breaks Rule 9-5-1
   {
      struct s u1;
      struct t u2;
   } a;
   a.u2.m3 = a.u1;   // Non-compliant
}
```

See also

Rule 9–5–1

6.0.3 Runtime failures

Rule 0–3–1	**(Document)**	**Minimization of run-time failures shall be ensured by the use of at least one of:** **(a) static analysis tools/techniques;** **(b) dynamic analysis tools/techniques;** **(c) explicit coding of checks to handle run-time faults.**

Rationale

Run-time checking is an issue (not specific to C++) to which developers need to pay special attention, especially as the C++ language is weak in its provision of any run-time checking. C++ implementations are not required to perform many of the dynamic checks that are necessary for robust software. It is therefore an issue that C++ developers need to consider carefully, adding dynamic checks to code wherever there is the potential for run-time errors to occur.

Where expressions consist only of values within a well-defined range, a run-time check may not be necessary, provided it can be demonstrated that for all values within the defined range the exception cannot occur. Such a demonstration, if used, should be documented along with the assumptions on which it depends. However, if adopting this approach, be very careful about subsequent modifications of the code that may invalidate the assumptions, or of the assumptions changing for any other reason.

The following notes give some guidance on areas where consideration needs to be given to the provision of dynamic checks.

- ***arithmetic errors***

  This includes errors occurring in the evaluation of expressions, such as overflow, underflow, division by zero or loss of significant bits through shifting.

  In considering integer overflow, note that unsigned integer calculations do not strictly overflow (producing undefined values), but the values wrap around (producing defined, but possibly unexpected, values).

- ***pointer arithmetic***

  Ensure that when an address is calculated dynamically the computed address is reasonable and points somewhere meaningful. In particular it should be ensured that if a pointer points within a structure or array, then when the pointer has been incremented or otherwise altered it still points to the same structure or array. See Rule 5–0–15, Rule 5–0–16, Rule 5–0–17 and Rule 5–0–18 for restrictions on pointer arithmetic.

- ***array bound errors***

  Ensure that array indices are within the bounds of the array size before using them to index the array.

- ***function arguments***

  Function arguments should be validated.

- ***pointer dereferencing***

  Where a function returns a pointer and that pointer is subsequently de-referenced the program should first check that the pointer is not *NULL*. Within a function, it is relatively straightforward to reason about which pointers may or may not hold *NULL* values. Across function boundaries, especially when calling functions defined in other source files or libraries, it is much more difficult.

```
// Given a pointer to a message, check the message header and return
// a pointer to the body of the message or NULL if the message is
// invalid.
const char_t *msg_body ( const char_t * msg )
{
   const char_t * body = NULL;
   if ( msg != NULL )
   {
      if ( msg_header_valid ( msg ) )
      {
         body = &msg [ MSG_HEADER_SIZE ];
      }
   }
   return ( body );
}

...
char_t  msg_buffer [ MAX_MSG_SIZE ];
const char_t * payload;
...
payload = msg_body ( msg_buffer );
if ( payload != NULL )
{
   // process the message payload
}
```

The techniques that will be employed to minimize run-time failures should be planned and documented, e.g. in design standards, test plans, static analysis configuration files, code review checklists.

Rule 0–3–2	(Required)	If a function generates error information, then that error information shall be tested.

Rationale

A function (whether it is part of the standard library, a third party library or a user defined function) may provide some means of indicating the occurrence of an error. This may be via a global error flag, a parametric error flag, a special return value or some other means. Whenever such a mechanism is provided by a function the calling program shall check for the indication of an error as soon as the function returns.

Note, however, that the checking of input values to functions is considered a more robust means of error prevention than trying to detect errors after the function has completed.

Example

```
extern void fn3 ( int32_t i, bool & flag );

int32_t fn1 ( int32_t i )
{
   int32_t result  = 0;
   bool  success = false;

   fn3 ( i, success );      // Non-compliant - success not checked
   return result;
}

int32_t fn2 ( int32_t i )
{
   int32_t result  = 0;
   bool  success = false;

   fn3 ( i, success );      // Compliant - success checked
   if ( !success )
   {
      throw 42;
   }
   return result;
}
```

See also

Rule 19–3–1

6.0.4 Arithmetic

Rule 0–4–1	**(Document)**	**Use of scaled-integer or fixed-point arithmetic shall be documented.**

Rationale

It is extremely difficult to design and implement arithmetic packages for scaled-integer or fixed-point arithmetic without overlooking dangerous cases.

If either is used, then this rule requires that documentation be produced to demonstrate that all the issues have been covered by the implementation.

Rule 0–4–2	**(Document)**	**Use of floating-point arithmetic shall be documented.**

Rationale

The safe use of floating-point arithmetic requires a high level of numerical analysis skills and in-depth knowledge of the compiler and target hardware.

If floating-point is to be used, then the following issues need to be covered as part of the deviation process:

- A justification explaining why floating-point is the appropriate or only solution.
- Demonstrate that appropriate skills are available.
- Demonstrate that an appropriate process is being applied.
- Document the floating-point implementation.

The paper "*What Every Computer Scientist Should Know about Floating-Point Arithmetic*" [13] explains the issues that need to be considered when using floating-point.

Example

When solving a quadratic equation, the value of $b^2–4.a.c$ is calculated. Assume that $a=1.22$, $b=3.34$, $c=2.28$ and that three significant digits are used during calculation. The exact value of $b^2–4.a.c$ is 0.0292. However, b^2 rounds to 11.2 and $4.a.c$ rounds to 11.1, giving a final answer of 0.1. This example demonstrates *catastrophic cancellation*. The subtraction does not cause an error, but it does expose errors introduced in the multiplications that are used when generating its operands.

Rule 0–4–3	**(Document)**	**Floating-point implementations shall comply with a defined floating-point standard.**

Rationale

Floating-point arithmetic has a range of problems associated with it. Some of these can be overcome by using an implementation that conforms to a recognized standard. An example of an appropriate standard is ANSI/IEEE Std 754 [14].

The definition of the floating-point types, in accordance with Rule 3–9–2, provides an opportunity for noting the floating-point standard in use, for example:

```
// IEEE 754 single-precision floating-point
typedef float float32_t;
```

6.1 General

6.1.0 Language

Rule 1–0–1	**(Required)**	**All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".**

[MISRA Guidelines Table 3; IEC 61508 Part 7: Table C.1]

Rationale

The MISRA C++ subset is based on ISO/IEC 14882:2003 [1]. No claim is made as to their suitability with respect to any other version of the standard. Any reference in this document to "C++" refers to the ISO/IEC 14882:2003 [1] standard.

It is recognized that it will be necessary to raise deviations (as described in Section 4.3.2) to permit certain language extensions, for example to support hardware specific features.

Deviations are required if the environmental limits, as specified in Annex B of ISO/IEC 14882:2003 [1], are exceeded.

Rule 1–0–2	**(Document)**	**Multiple compilers shall only be used if they have a common, defined interface.**

[Implementation 7.5(1, 2, 9)]

Rationale

"Multiple compilers" includes:

- Mixed languages;
- Different compilers;
- Different versions of the same compiler;
- Different configurations of the same compiler;

where "compiler" includes any tool used to translate source code or link object code.

If a module is to be implemented in a language other than C++, or compiled using a different C++ compiler, then it is essential to ensure that the module will integrate correctly with other modules. Some aspects of the behaviour of the C++ language are *implementation-defined*, and therefore these must be understood for the compiler being used. Examples of issues that need to be understood include:

- Stack usage;
- Parameter passing;
- The way in which data values are stored (lengths, alignments, aliasing, overlays etc.).

Note that this includes the use of `extern "C"`.

Rule 1–0–3	(Document)	The implementation of integer division in the chosen compiler shall be determined and documented.

[Implementation 5.6(4)]

Rationale

An ISO compliant compiler can do one of two things when dividing two signed integers, one of which is positive and one negative. Firstly, it may round up with a negative remainder (e.g. -5/3 = -1 remainder -2) or, secondly, it may round down with a positive remainder (e.g. -5/3 = -2 remainder +1).

It is important to determine which of these is implemented by the compiler and to document it for developers.

Note that this rule also covers modulus as it is defined in terms of division.

6.2 Lexical conventions

6.2.2 Character sets

Rule 2–2–1	(Document)	The character set and the corresponding encoding shall be documented.

Rationale

The source code is written in one or more character sets. Optionally, the program can execute in a further or multiple character sets. All the character sets that are used shall be documented.

Documenting the character sets that are used increases developer awareness, preventing issues arising due to the use of incompatible character sets.

For example, ISO 10646-1 [15] defines an international standard for mapping character sets to numeric values. For portability, *character-constants* and *string-literals* should only contain characters that map to a documented subset.

6.2.3 Trigraph sequences

Rule 2–3–1	(Required)	*Trigraphs* shall not be used.

Rationale

Trigraphs are denoted by a sequence of 2 question marks followed by a specified third character (e.g. `??-` represents a "~" (tilde) character and `??)` represents a "]"). They can cause accidental confusion with other uses of two question marks.

Example

The string

```
"(Date should be in the form ??-??-??)"
```

would probably not meet developer expectations, since the compiler would interpret it as

```
"(Date should be in the form ~~]"
```

6.2.5 Alternative tokens

Rule 2–5–1	(Advisory)	*Digraphs* should not be used.

Rationale

The *digraphs* are:

```
<%  %>  <:  :>  %:  %:%:
```

The use of *digraphs* may not meet developer expectations.

Example

```
template < typename T>
class A
{
   public:
      template < int32_t i >
      void f2 ( );
};
void f ( A<int32_t> * a<:10:> )      // Non-compliant
<% a<:0:>->f2<20> ( ); %>            // Non-compliant

// The above is equivalent to:
void f ( A<int32_t> * a[ 10 ] )
{
   a[ 0 ]->f2<20> ( );               // Compliant
}
```

6.2.7 Comments

Rule 2–7–1	(Required)	The character sequence `/*` shall not be used within a C-style comment.

Rationale

C++ does not support the nesting of C-style comments even though some compilers support this as a non-portable language extension. A comment beginning with `/*` continues until the first `*/` is encountered. Any `/*` occurring inside a comment is a violation of this rule.

Example

Consider the following code fragment:

```
/* some comment, end comment marker accidentally omitted
Perform_Critical_Safety_Function(X);
/* this "comment" is Non-compliant */
```

In reviewing the code containing the call to the function, the assumption is that it is executed code.

Because of the accidental omission of the end comment marker, the call to `Perform_Critical_Safety_Function` will not be executed.

Rule 2–7–2	(Required)	Sections of code shall not be “commented out” using C-style comments.

Rationale

Using C-style start and end comment markers for this purpose is dangerous because C-style comments do not support nesting, and any comments already existing in the section of code would change the effect.

Additionally, comments should only be used to explain aspects of the code that may not be clear from the source code itself. Code that is commented-out may become out of date, which may lead to confusion when maintaining the code.

A more appropriate method of recording the history of changes in source code (e.g. a Source Control System) should be used instead of commenting-out.

Example

```
void fn ( int32_t i )
{
   /*
      ++i;      /* We want to increment "i" */
   */
   for ( int32_t j = 0 ; j != i ; ++j )
   {
   }
}
```

See also

Rule 2–7–1, Rule 2–7–3

Rule 2–7–3	(Advisory)	Sections of code should not be “commented out” using C++ comments.

Rationale

Ideally, comments should only be used to explain aspects of the code that may not be clear from the source code itself. Code that is commented-out may become out of date, which may lead to confusion when maintaining the code.

A more appropriate method of recording the history of changes in source code (e.g. a Source Control System) should be used instead of commenting-out.

Example

```
void fn ( int32_t i )
{
   // ++i;   // We want to increment
   for ( int32_t j = 0 ; j != i ; ++j )
   {
   }
}
```

See also

Rule 2–7–2

6.2.10 Identifiers

Rule 2–10–1	(Required)	Different identifiers shall be typographically unambiguous.

Rationale

Depending on the font used to display the character set, it is possible for certain glyphs to appear the same, even though the characters are different. This may lead to the developer confusing an identifier with another one.

To help reduce the chance of this, identifiers shall not differ by any combination of:

- Only a mixture of case;
- The presence or absence of the underscore character;
- The interchange of the letter “O”, and the number “0”;
- The interchange of the letter “I”, and the number “1”;
- The interchange of the letter “I”, and the letter “l” (el);
- The interchange of the letter “l” (el), and the number “1”;
- The interchange of the letter “S” and the number “5”;
- The interchange of the letter “Z” and the number “2”;
- The interchange of the letter “n” and the letter “h”;
- The interchange of the letter “B” and the number “8”;
- The interchange of the letter sequence “rn” (“r” followed by “n”) with the letter “m”.

Example

```
int32_t id1_a_b_c;
int32_t id1_abc;      // Non-compliant

int32_t id2_abc;      // Non-compliant
int32_t id2_ABC;      // Non-compliant

int32_t id3_a_bc;
int32_t id3_ab_c;     // Non-compliant

int32_t id4_a_bc;
int32_t id4_ab_c;     // Non-compliant

int32_t id5_ii;
int32_t id5_11;       // Non-compliant

int32_t id6_i0;
int32_t id6_1O;       // Non-compliant

int32_t id7_in;
int32_t id7_1h;       // Non-compliant

int32_t id8_Z5;
int32_t id8_2S;       // Non-compliant

int32_t id9_ZS;
int32_t id9_25;       // Non-compliant
```

Rule 2–10–2	(Required)	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

If an identifier is declared in an inner scope and it uses the same name as an identifier that already exists in an outer scope, then the innermost declaration will "hide" the outer one. This may lead to developer confusion.

The terms outer and inner scope are defined as follows:

- Identifiers that have file scope can be considered as having the outermost scope.
- Identifiers that have block scope have a more inner scope.
- Successive, nested blocks, introduce more inner scopes.

Example

```
int16_t i;
{
   int16_t i;   // This is a different variable
                // This is Non-compliant
   i = 3;       // It could be confusing as to which i this refers
}
void fn ( int16_t i )   // Non-compliant
{
}
```

Rule 2–10–3	(Required)	A *typedef* name (including qualification, if any) shall be a *unique* identifier.

Rationale

Reusing a *typedef* name either as another *typedef* name or for any other purpose may lead to developer confusion.

The same *typedef* shall not be duplicated anywhere in the *project*, even if the declarations are identical.

Note that where the type definition is made in a *header file*, and that *header file* is included in multiple source files, this rule is not violated.

Example

```
// f1.cc
namespace NS1
{
   typedef int16_t WIDTH;
}
// f2.cc
namespace NS2
{
   float32_t WIDTH;   // Compliant -
                      //        NS2::WIDTH is not the same as NS1::WIDTH
}
```

```
void f1 ( )
{
   typedef int32_t TYPE;
}
void f2 ( )
{
   float32_t TYPE;   // Non-compliant
}
```

Rule 2–10–4	(Required)	A *class, union* or *enum* name (including qualification, if any) shall be a *unique* identifier.

Rationale

Reusing a *class*, *union* or *enum* name, either as another type or for any other purpose, may lead to developer confusion.

The *class*, *union* or *enum* name shall not be duplicated anywhere in the *project*, even if the declarations are identical.

This rule is not violated when the definition is made in a *header file*, and that *header file* is included in multiple source files.

Example

```
void f1 ( )
{
   class TYPE { };
}
void f2 ( )
{
   float32_t TYPE;  // Non-compliant
}
```

Rule 2–10–5	(Advisory)	The identifier name of a non-member object or function with static storage duration should not be reused.

Rationale

Regardless of scope, no identifier with static storage duration should be re-used across any source files in the *project*. This includes objects or functions with external linkage and any objects or functions with the static storage class specifier.

While the compiler can understand this and is in no way confused, the possibility exists for the developer to incorrectly associate unrelated variables with the same name.

Example

```
namespace NS1
{
   static int32_t global = 0;
}
```

```
namespace NS2
{
   void fn ( )
   {
      int32_t global;      // Non-compliant
   }
}
```

Rule 2–10–6	(Required)	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Rationale

For C compatibility, it is possible in C++ for a name to refer to both a type and object or a type and function. This can lead to confusion.

Example

```
typedef struct vector { uint16_t x ; uint16_t y; uint16_t z; } vector;
// Non-compliant ^^                                  Non-compliant ^^
        struct vector { uint16_t x ; uint16_t y; uint16_t z; } vector;
// Non-compliant ^^                                  Non-compliant ^^
```

6.2.13 Literals

Rule 2–13–1	(Required)	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

[Undefined 2.13.2(3)]

Rationale

The use of an undefined escape sequence leads to *undefined behaviour*.

The defined escape sequences (ISO/IEC 14882:2003 [1] §2.13.2) are:

\n, \t, \v, \b, \r, \f, \a, \\, \?, \', \", \<*Octal Number*>, \x<*Hexadecimal Number*>

Example

```
void fn ( )
{
   const char_t a[ 2 ] = "\k";      // Non-compliant
   const char_t b[ 2 ] = "\b";      // Compliant
}
```

See also

Rule 2–13–2

Rule 2–13–2	(Required)	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

[Implementation 2.13.2(1, 2)]

Rationale

Any integer constant beginning with a "0" (zero) is treated as octal. Because of this, it is possible for a zero-prefixed constant that is intended to be a decimal number to be incorrectly entered as an octal number, contrary to developer expectations.

Octal escape sequences can also be problematic because the inadvertent introduction of a decimal digit (i.e. "8" or "9") ends the octal escape and introduces another character.

The integer constant zero (written as a single numeric digit), is strictly speaking an octal constant, but is a permitted exception to this rule. Additionally, "\0" is the only permitted octal escape sequence.

Example

The following array initialization for 3-digit decimal bus messages would not behave as expected:

```
code[ 1 ] = 109;   // Compliant     - decimal 109
code[ 2 ] = 100;   // Compliant     - decimal 100
code[ 3 ] = 052;   // Non-compliant - equivalent to decimal 42
code[ 4 ] = 071;   // Non-compliant - equivalent to decimal 57
```

The value of the first expression in the following example is *implementation-defined* because the character constant consists of two characters, "\10" and "9". The second character constant expression contains the single character "\100".

```
code[ 5 ] = '\109';   // Non-compliant - implementation-defined,
                      //                 two character constant
code[ 6 ] = '\100';   // Non-compliant - set to 64.
```

Rule 2–13–3	(Required)	A "*U*" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Rationale

The type of an integer is dependent on a complex combination of factors including:

- The magnitude of the constant;
- The implemented sizes of the integer types;
- The presence of any suffixes;
- The number base in which the value is expressed (i.e. decimal, octal or hexadecimal).

For example, the value 0x8000 is of type *unsigned int* in a 16-bit environment, but of type (*signed*) *int* in a 32-bit environment. If an overload set includes candidates for an *unsigned int* and an *int*, then the overload that would be matched by 0x8000 is therefore dependent on the implemented integer size. Adding a "*U*" suffix to the value specifies that it is unsigned.

Note that the usage context may also require the use of suffixes, as shown in Section 6.5.0.

Example

```
template <typename T>
void f ( T );

template <>
void f < uint16_t > ( uint16_t );

template <>
void f < int16_t > ( int16_t );

void b ( )
{
   uint16_t u16a = 0U;     // Compliant

   f ( 0x8000 );           // Non-compliant on a 16-bit platform.
   u16a = u16a + 0x8000;   // Non-compliant as context is unsigned.
}
```

Rule 2–13–4 (Required) Literal suffixes shall be upper case.

Rationale

Using upper case literal suffixes removes the potential ambiguity between “1” (digit 1) and “l” (letter el) for declaring literals.

Example

```
const uint32_t   a = 0U;
const uint32_t   b = 0u;        // Non-compliant
const int64_t    c = 0L;
const int64_t    d = 0l;        // Non-compliant
const uint64_t   e = 0UL;
const uint64_t   f = 0Ul;       // Non-compliant
const uint32_t   g = 0x12bU;
const uint32_t   h = 0x12bu;    // Non-compliant
const float32_t  m = 1.2F;
const float32_t  n = 2.4f;      // Non-compliant
const float128_t p = 1.2L;
const float128_t n = 2.4l;      // Non-compliant
```

See also

ISO/IEC 14882:2003 [1] §2.13

Rule 2–13–5 (Required) Narrow and wide string literals shall not be concatenated.

[Undefined 2.13.4(3)]

Rationale

Concatenation of wide and narrow string literals leads to *undefined behaviour*.

Example

```
char_t  n_array[] =  "Hello"  "World";   // Compliant
wchar_t w_array[] = L"Hello" L"World";   // Compliant
wchar_t mixed[]   =  "Hello" L"World";   // Non-compliant
```

6.3 Basic concepts

6.3.1 Declarations and definitions

Rule 3–1–1	(Required)	It shall be possible to include any *header file* in multiple translation units without violating the *One Definition Rule*.

Rationale

Header files should be used to declare objects, functions, inline functions, function templates, *typedefs*, macros, classes, and class templates and shall not contain or produce definitions of objects or functions (or fragment of functions or objects) that occupy storage.

A *header file* is considered to be any file that is included via the *#include* directive, regardless of name or suffix.

Example

```
// a.h
       void f1 ( );       // Compliant
       void f2 ( ) { }    // Non-compliant
inline void f3 ( ) { }    // Compliant
template <typename T>
void f4 ( T ) { }         // Compliant

int32_t a;                // Non-compliant

// a.cpp
#include "a.h"
```

Rule 3–1–2	(Required)	Functions shall not be declared at block scope.

Rationale

A function declared at block scope will refer to a member of the enclosing namespace, and so the declaration should be explicitly placed at the namespace level.

Additionally, where a declaration statement could either declare a function or an object, the compiler will choose to declare the function. To avoid potential developer confusion over the meaning of a declaration, functions should not be declared at block scope.

Example

```
class A
{
};

void b1 ( )
{
   void f1 ();   // Non-compliant - declaring a function in block scope
   A a ();       // Non-compliant - appears to declare an object with no
                 // arguments to constructor, but it too declares a
                 // function 'a' returning type 'A' and taking no
                 // parameters.
}
```

Rule 3–1–3	(Required)	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

[Undefined 5.7(5, 6)]

Rationale

Although it is possible to declare an array of incomplete type and access its elements, it is safer to do so when the size of the array can be explicitly determined.

Example

```
       int32_t array1[ 10 ];               // Compliant
extern int32_t array2[ ];                  // Non-compliant
       int32_t array3[ ] = { 0, 10, 15 };  // Compliant
extern int32_t array4[ 42 ];               // Compliant
```

6.3.2 One Definition Rule

The *One Definition Rule* is defined in Section 3.2 of ISO/IEC 14882:2003 [1].

In essence, the requirement for *One Definition Rule* arises because C++ compilers treat each source file (with included headers) as separate "translation units" where each translation unit is compiled in isolation. The set of compiled translation units are then linked to form the executable program.

The linker is allowed to assume that objects, templates, types, etc. that share the same name in different translation units refer to the same definition. The linker is not required to check that these definitions are the same.

In the following example the same `struct S` appears to be defined in both translation units, but as the definitions are not the same, the result is not what the developer expects.

```
// source file file1.cpp
struct S
{
   int32_t x;
   int32_t y;
};
int32_t XminusY ( S & s )
{
   return ( s.x - s.y );
}
// source file file2.cpp
struct S {
   int32_t y;   // note order of x and y exchanged
   int32_t x;
};
void setX ( S & s, int32_t v ) { s.x = v; }
void setY ( S & s, int32_t v ) { s.y = v; }
```

The user may be surprised that the result of `XminusY` is `y - x` not `x - y`.

As stated above, the linker is not required to check the compatibility of the two definitions; the *One Definition Rule* puts the onus on the developer to ensure that the definitions are compatible. The following rules (Rule 3–2–1 to Rule 3–2–4) reinforce the need to follow the *One Definition Rule*, and provide specific instructions for the developer.

Rule 3–2–1	(Required)	All declarations of an object or function shall have *compatible types*.

[NDR 3.2(3), Undefined 3.2(5)]

Rationale

It is *undefined behaviour* if the declarations of an object or function in two different translation units do not have *compatible types*.

The easiest way of ensuring object or function types are compatible is to make the declarations identical.

Example

```
// File a.cpp
extern int32_t a;
extern int32_t b [];
extern char_t  c;

int32_t f1 ( );
int32_t f2 ( int32_t );

// File b.cpp
extern int64_t a;           // Non-compliant – not compatible
extern int32_t b [ 5 ];     // Compliant
int16_t c;                  // Non-compliant

char_t f1 ( );              // Non-compliant
char_t f2 ( char_t );       // Compliant – not the same function as
                            //             int32_t f2 ( int32_t )
```

See also

Rule 3–9–1

Rule 3–2–2	(Required)	The *One Definition Rule* shall not be violated.

Rationale

Violation of the *One Definition Rule* ([1] §3.2) leads to *undefined behaviour*. In general, this means that the program shall contain exactly one definition of every non-inline function or object.

Additionally:

- The definitions of a type shall consist of the same sequence of tokens, and;
- The definitions of a template shall consist of the same sequence of tokens, and;
- The definitions of an inline function shall consist of the same sequence of tokens.

Note that for the purposes of this rule, *typedefs* shall be treated as types.

Example

```
// File a.cpp
struct S1
{
   int32_t i;
};
```

```
struct S2
{
   int32_t i;
};
// File b.cpp
struct S1
{
   int64_t i;
};                // Non-compliant – token sequence different
struct S2
{
   int32_t i;
   int32_t j;
};                // Non-compliant - token sequence different
```

Rule 3–2–3	(Required)	A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Rationale

Having a single declaration of a type, object or function allows the compiler to detect incompatible types for the same entity.

Normally, this will mean declaring an external identifier in a *header file* that will be included in any file where the identifier is defined or used.

Example

```
// header.hpp
extern int16_t a;
// file1.cpp
#include "header.hpp"
extern int16_t b;
// file2.cpp
#include "header.hpp"
extern int32_t b;   // Non-compliant - compiler may not detect the error
int32_t a;          // Compliant     - compiler will detect the error
```

Rule 3–2–4	(Required)	An identifier with external linkage shall have exactly one definition.

[NDR 3.2(3), Undefined 3.2(5)]

Rationale

It is *undefined behaviour* if an identifier is used for which multiple definitions exist (in different translation units) or no definition exists at all. With the exception of templates and inline functions, multiple definitions in different translation units are not permitted, even if the definitions are the same.

Example

```
// file1.cpp
int32_t i = 0;

// file2.cpp
int32_t i = 1;      // Non-compliant
```

6.3.3 Declarative regions and scope

Rule 3–3–1	**(Required)**	**Objects or functions with external linkage shall be declared in a *header file*.**

Rationale

Placing the declarations of objects and functions with external linkage in a *header file* documents that they are intended to be accessible from other translation units.

If external linkage is not required, then the object or function shall either be declared in an unnamed namespace or declared *static*.

This will reduce the visibility of objects and functions, which is considered to be good practice.

Exception

This rule does not apply to *main*, or to members of unnamed namespaces.

Example

```
// header.hpp
extern int32_t a1;
extern void f3 ( );

// file1.cpp
#include "header.hpp"
       int32_t a1 = 0;    // Compliant
       int32_t a2 = 0;    // Non-compliant
static int32_t a3 = 0;    // Compliant

namespace
{
   int32_t a4 = 0;        // Compliant

   void f1 ()             // Compliant
   {
   }
}

static void f2 ( )        // Compliant
{
}

void f3 ( )               // Compliant
{
   a1 = 1;
   a2 = 1;
   a3 = 1;
   a4 = 1;
}
```

```
void f4 ( )           // Non-compliant
{
   a1 = 2;
   a2 = 2;
   a3 = 2;
   a4 = 2;
}
void main ( )         // Compliant by exception
{
   f1 ( );
   f2 ( );
   f3 ( );
   f4 ( );
}
```

Rule 3–3–2	(Required)	If a function has internal linkage then all re-declarations shall include the *static* storage class specifier.

Rationale

If the declaration of a function includes the *static* storage class specifier, then it has internal linkage.

A re-declaration of such a function is not required to have the *static* keyword, but it will still have internal linkage. However, this is implicit and may not be obvious to a developer. It is therefore good practice to apply the *static* keyword consistently so that the linkage is explicitly stated.

Example

```
static void f1 ( );
static void f2 ( );
       void f1 ( ) { }      // Non-compliant
static void f2 ( ) { }      // Compliant
```

6.3.4 Name lookup

Rule 3–4–1	(Required)	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Rationale

Defining variables in the minimum block scope possible reduces the visibility of those variables and therefore reduces the possibility that these identifiers will be used accidentally. A corollary of this is that global objects (including singleton function objects) shall be used in more than one function.

Example

```
void f ( int32_t k )
{
   int32_t j = k * k;      // Non-compliant
   {
      int32_t i = j;       // Compliant
      std::cout << i << j << std::endl;
   }
}
```

In the above example, the definition of `j` could be moved into the same block as `i`, reducing the possibility that `j` will be incorrectly used later in `f`.

6.3.9 Types

Rule 3–9–1	**(Required)**	**The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.**

Rationale

If a re-declaration has *compatible types* but not types which are token-for-token identical, it may not be clear to which declaration that re-declaration refers.

Example

```
typedef int32_t INT;
        INT     i;
extern int32_t i;         // Non-compliant
        INT j;
extern INT j;             // Compliant
// The following lines break Rule 3-9-2
extern void f ( signed int );
        void f (        int );         // Non-compliant
extern void g ( const  int );
        void g (        int );         // Non-compliant
```

See also

Rule 3–2–1, Rule 3–9–2

Rule 3–9–2	**(Advisory)**	***typedefs* that indicate size and signedness should be used in place of the basic numerical types.**

[Implementation 3.9.1(1, 5)]

Rationale

The basic numerical types of *char*, *int*, *short*, *long*, *float*, *double* and *long double* should not be used, but specific-length *typedefs* should be used. This rule helps to clarify the size of the storage, but does not guarantee portability because of the asymmetric behaviour of integral promotion. See

the discussion of integral promotion in Section 6.5.0. It is still important to understand the integer size of the implementation.

Developers should be aware of the actual implementation of the *typedefs* under these definitions.

Exception

The *wchar_t* does not need a *typedef* as it always maps to a type that supports wide characters.

The *char_t typedef* does not indicate size and signedness and is simply included to allow *char* objects to be declared without the use of the basic *char* type, allowing any use of (plain) *char* to be detected and reported by analysis tools.

Example

The ISO (POSIX) *typedefs* as shown below are recommended and are used for all basic numerical and character types in this document. For a 32-bit integer machine, these are as follows:

```
typedef          char   char_t;
typedef signed   char   int8_t;
typedef signed   short  int16_t;
typedef signed   int    int32_t;
typedef signed   long   int64_t;
typedef unsigned char   uint8_t;
typedef unsigned short  uint16_t;
typedef unsigned int    uint32_t;
typedef unsigned long   uint64_t;
typedef          float  float32_t;
typedef          double float64_t;
typedef long     double float128_t;
```

typedefs are not considered necessary in the specification of bit-field types.

Rule 3–9–3	**(Required)**	**The underlying bit representations of floating-point values shall not be used.**

[Implementation 3.9.1(8)]

Rationale

The storage layout used for floating-point values may vary from one compiler to another, and therefore no floating-point manipulations shall be made which rely directly on the way the values are stored. The in-built operators and functions, which hide the storage details from the developer, should be used.

Example

```
float32_t My_fabs ( float32_t f )
{
   uint8_t * pB = reinterpret_cast< uint8_t * >( &f );
   *( pB + 3 ) &= 0x7f;   // Non-compliant – generate the absolute value
                          //                 of an IEEE-754 float value.
   return ( f );
}
```

6.4 Standard conversions

6.4.5 Integral promotions

Rule 4–5–1	**(Required)**	**Expressions with type *bool* shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.**

Rationale

The use of *bool* operands with other operators is unlikely to be meaningful (or intended). This rule allows the detection of such uses, which often occur because the logical operators (`&&`, `||` and `!`) can be easily confused with the bitwise operators (`&`, `|` and `~`).

Example

```
bool   b1  = true;
bool   b2  = false;
int8_t s8a;

if ( b1 & b2 )     // Non-compliant
if ( b1 < b2 )     // Non-compliant
if ( ~b1 )         // Non-compliant
if ( b1 ^ b2 )     // Non-compliant
if ( b1 == false ) // Compliant
if ( b1 == b2 )    // Compliant
if ( b1 != b2 )    // Compliant
if ( b1 && b2 )    // Compliant
if ( !b1 )         // Compliant
s8a = b1 ? 3 : 7;  // Compliant
```

Rule 4–5–2	**(Required)**	**Expressions with type *enum* shall not be used as operands to built-in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.**

Rationale

Enumerations have *implementation-defined* representation and so should not be used in arithmetic contexts.

Example

```
enum { COLOUR_0, COLOUR_1, COLOUR_2, COLOUR_COUNT } colour;

if ( COLOUR_0 == colour )                 // Compliant
if ( ( COLOUR_0 + COLOUR_1 ) == colour )  // Non-compliant
if ( colour < COLOUR_COUNT )              // Compliant
```

Rule 4–5–3	(Required)	**Expressions with type (plain) *char* and *wchar_t* shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.**

Rationale

Manipulation of character data may generate results that are contrary to developer expectations. For example, ISO/IEC 14882:2003 [1] §2.2(3) only requires that the digits "0" to "9" have consecutive numerical values.

Exception

Exceptionally, the following operators may be used if the associated restriction is observed:

- The binary + operator may be used to add an integral value in the range 0 to 9 to '0';
- The binary – operator may be used to subtract character '0';
- The relational operators <, <=, >, >= may be used to determine if a character (or wide character) represents a digit.

Example

```
char_t  ch = 't';                       // Compliant
uint8_t v;

if ( ( ch >= 'a' ) && ( ch <= 'z' ) ) // Non-compliant
{
}

if ( ( ch >= '0' ) && ( ch <= '9' ) ) // Compliant by exception
{
   v = ch – '0';                        // Compliant by exception
   v = ch – '1';                        // Non-compliant
}

else
{
   // ...
}
ch = '0' + v;                           // Compliant by exception
ch = 'A' + v;                           // Non-compliant
```

6.4.10 Pointer conversions

Rule 4–10–1	(Required)	*NULL* shall not be used as an integer value.

Rationale

In C++, the literal 0 is both an integer type and the *null-pointer-constant*. To meet developer expectations, *NULL* should be used as the *null-pointer-constant*, and 0 for the integer zero.

Note: as a result of this rule, *NULL* is considered to have pointer type.

Example

```
#include <cstddef>

void f1 ( int32_t   );
void f2 ( int32_t * );

void f3 ( )
{
   f1 ( NULL );   // Not-compliant, NULL used as an integer
   f2 ( NULL );   // Compliant
}
```

Rule 4–10–2	(Required)	Literal zero (0) shall not be used as the *null-pointer-constant*.

Rationale

In C++, the literal 0 is both an integer type and the *null-pointer-constant*. To meet developer expectations, *NULL* should be used as the *null-pointer-constant*, and 0 for the integer zero.

Example

```
#include <cstddef>

void f1 ( int32_t   );
void f2 ( int32_t * );

void f3 ( )
{
   f1 ( 0 );      // Compliant
   f2 ( 0 );      // Non-compliant, 0 used as the null pointer constant
}
```

6.5 Expressions

6.5.0 General

Strong typing

When developing critical systems it is considered best practice to use strong typing. This facility is not rigidly enforced for the built-in types of the C++ language, so the following guidance has been produced to strengthen the use of those types.

Arithmetic type conversions

Implicit and explicit type conversions

The C++ language allows the developer considerable freedom and will allow conversions between different arithmetic types to be performed automatically. An explicit cast may be introduced for functional reasons, for example:

- To change the type in which a subsequent arithmetic operation is performed.
- To truncate a value deliberately.
- To make a type conversion explicit in the interests of clarity.

The insertion of a cast for purposes of clarification is often helpful, but when taken to excess, the practice can lead to unreadable code. As demonstrated below, there are some implicit conversions that can safely be ignored and others that cannot.

Types of implicit conversion

There are three particular categories of implicit type conversion that need to be distinguished.

Integral promotion conversions

Integral promotion describes a process whereby arithmetic operations are always conducted on integer operands of type *int* or *long* (*signed* or *unsigned*). Operands of type *char*, *short* and *bool* are always converted to type *int* or *unsigned int* whilst those of type *wchar_t* and *enum* may be converted to *int*, *unsigned int*, *long* or *unsigned long*. These are referred to as *small integer types*.

The rules of integral promotion ([1] §4.5) require that in most arithmetic operations, an operand of a *small integer type* be converted to *int*, if an *int* is able to represent all values of the original type; otherwise the value is converted to *unsigned int*, *long* or *unsigned long*. A *bit-field* will be converted to an *int* if an *int* is able to represent all the values; it will be converted to an *unsigned int* if an *unsigned int* is able to represent all the values, otherwise no promotion takes place.

Integral promotion is frequently confused with “balancing” of operands (described below). Balancing only occurs for some binary operators, whilst integral promotion can take place for other expressions when a *small integer* is used.

Because of integral promotion, the result of adding two objects of type *unsigned short* is always a value of type *signed int* or *unsigned int*; in fact, the addition is performed in this promoted type. It is therefore possible for such an operation to derive a result whose value exceeds the size that could be accommodated in the original type of the operands. For example, if the size of an *int* is 32 bits, it is possible to multiply two objects of type *short* (16 bits) and derive a 32-bit result with no danger of overflow. On the other hand if the size of an *int* is only 16 bits, the product of two 16-bit objects will only yield a 16-bit result and care must be taken to ensure that an unexpected

overflow does not remain undetected.

Integral promotion also applies to some unary operators. For example, the result of applying a bitwise negation operator (~) to an *unsigned char* operand is typically a negative value of type *signed int.*

Integral promotion is a fundamental inconsistency in the C++ language (inherited from C) whereby the *small integer types* behave differently from *long* and *int* types. The use of *typedefs* is a practice that is encouraged in MISRA C++. However, because the behaviour of the various integer types is not consistent, it can be unsafe to ignore the underlying base types (see description on following pages) unless some restrictions are placed on the way in which expressions are constructed. It is the intention of the following rules that the effects of integral promotion should be neutralized in order to avoid these anomalies.

Assigning conversions

Assigning conversions occur when:

- The type of an assignment expression is converted to the type of the assignment object;
- The type of an initializer expression is converted to the type of the initialized object;
- The type of a function call argument is converted to the type of the formal parameter as declared in the function prototype;
- The type of the expression used in a return statement is converted to the type of the function as declared in the function prototype;
- The type of the constant expression in a switch case label is converted to the promoted type of the controlling expression. This conversion is performed only for the purposes of comparison.

In each case, the value of an arithmetic expression is unconditionally converted, where necessary, to another type.

Balancing conversions

Balancing conversions for arithmetic and enumeration types are described in ISO/IEC 14882:2003 ([1] §5(9)) under the term “Usual Arithmetic Conversions”. This is a set of rules that provides a mechanism to yield a common type when two operands of a binary operator are balanced to a common type. Also, the second and third arguments of the conditional operator (`? :`) ([1] §5.16) are sometimes balanced to a common type.

The balancing rules are preceded by the process of integral promotion (described above). Integral promotion happens as part of the usual arithmetic conversions even when two operands are of identical type.

Integral types having special semantics

In C++ there are a number of types that have an integral representation and can be freely used in any arithmetic expression. The types concerned are *bool*, *char*, *wchar_t* and *enum*. For the purpose of this document, *bit-field* objects are considered to have a special type.

Thus, an expression of type *bool* can be multiplied by an expression of type *char,* which is then added to an expression of *enum* type.

Terminology will be defined and rules introduced to restrict the use of these types to appropriate contexts.

Bit-fields

Depending on the width of a *bit-field*, it may undergo integral promotions and so, like the *small integer types*, it can lead to operations that have dangerous conversions. Therefore *bit-fields* should be considered as integral types of fixed length (that of the number of specified bits) the permissible range being from length zero to the number of bits of the largest integral type.

Dangerous type conversions

There are a number of potential dangers associated with type conversions that must be avoided:

- **Loss of value**: Conversion to a type where the magnitude of the value cannot be represented.
- **Loss of sign**: Conversion from a signed type to an unsigned type resulting in loss of sign.
- **Loss of precision**: Conversion from a floating type to an integer type with consequent loss of precision. Conversion from a floating type to a narrower floating type (which may also be a loss of value).

The only type conversions that can be guaranteed safe for all data values and all possible conforming implementations are:

- Conversion of an integral value to a wider type of the same signedness;
- Conversion of a floating type to a wider floating type.

In practice, if assumptions are made about typical type sizes, it is possible to classify other type conversions as safe. In general, MISRA C++ adopts the principle that it is wise to identify potentially dangerous type conversions by requiring that the conversion be made explicit.

Other dangers in the area of type conversion also need to be recognized. These issues arise from areas of misunderstanding and difficulty in the C++ language rather than because data values are not preserved.

- **Type widening in integral promotion**: The type in which integral expressions are evaluated depends on the type of the operands after any integral promotion. It is always possible to multiply two 8-bit values and access a 16-bit result if the magnitude requires it. It is sometimes, though not always, possible to multiply two 16-bit values and retrieve a 32-bit result. It is safer never to rely on the widening type afforded by integral promotion. Consider the following example:

```
uint16_t u16a = 40000;          // unsigned short / unsigned int ?
uint16_t u16b = 30000;          // unsigned short / unsigned int ?
uint32_t u32x = u16a + u16b;    // u32x = 70000 or 4464 ?
```

  The expected result is presumably 70000, but the value assigned to `u32x` will in practice depend on the implemented size of an *int*. If the implemented size of an *int* is 32 bits, the addition will occur in 32-bit signed arithmetic and the correct value will be stored. If the implemented size of an *int* is only 16 bits, the addition will take place in 16-bit unsigned arithmetic, wraparound will occur and will yield the value 4464 (70000 % 65536). Wraparound in unsigned arithmetic is well defined and may even be intended, but is potentially confusing.

- **Evaluation type confusion**: A similar problem arises from a common misconception among developers that the type in which a calculation is conducted is influenced in some way by the type to which the result is assigned or converted. For example, in the following

code the two 16-bit objects are added together in 16-bit arithmetic (on a platform with a 16-bit *int*), and the result is converted to type *uint32_t* on assignment.

```
u32x = u16a + u16b;
```

It is not unusual for developers to be deceived into thinking that the addition is performed in 32-bit arithmetic — because of the type of `u32x`.

Confusion of this nature is neither confined to integer arithmetic nor to implicit conversions. The following examples demonstrate some statements in which the result is well defined but the calculation may not be performed in the type that the developer assumes.

```
u32a = static_cast< uint32_t >(u16a * u16b);  // Evaluated in u16
f64a = u16a / u16b;                           // Evaluated in u16
f32a = static_cast< float32_t >(u16a / u16b); // Evaluated in u16
f64a = f32a + f32b;                           // Evaluated in f32
f64a = static_cast< float64_t >(f32a + f32b); // Evaluated in f32
```

- **Change of signedness in arithmetic operations**: Integral promotion will often result in two unsigned operands yielding a result of type *(signed) int*. For example, the addition of two 16-bit unsigned operands will yield a signed 32-bit result if *int* is 32 bits, but an unsigned 16-bit result if *int* is 16 bits.
- **Change of signedness in bitwise operations**: Integral promotion can have some particularly unfortunate repercussions when bitwise operators are applied to small unsigned types. For example, a bitwise complement operation on an operand of type *unsigned char* will generally yield a result of type *(signed) int* with a negative value. The operand is promoted to type *int* before the operation and the extra high order bits are set by the complement process. The number of extra bits, if any, is dependent on the size of an *int*, which is hazardous if the complement operation is followed by a right shift.

In order to avoid the pitfalls associated with the issues described above, it is important to establish some principles to constrain the way in which expressions are constructed. To start with, definitions of some concepts are presented below.

Underlying type

In a programming language, a type has a representation that is a mapping onto the hardware architecture. A type also has an implementation equating to the set of operations which can be performed on that type via its representation.

MISRA C:2004 introduced the concept of *underlying type*. Unfortunately the term is already used in C++ as a euphemism for representation, where the representation of one type is described in terms of other types. This use, however, is confined to the *wchar_t* and enumeration types (see Sections 3.9.1(5) and 7.2(6) of ISO/IEC 14882:2003 [1]). In both cases, from the MISRA point of view, the *underlying type* of these types will be simple and hence the possibility of confusion will be minimal. Consequently, it is appropriate to adopt the MISRA C:2004 concept and terminology.

Since C++ has an explicit *bool* type it is not considered necessary to permit the “Boolean by construction” concept introduced in MISRA C:2004.

Underlying type is a conceptual departure from the C++ language in which integral promotion does not exist, and the usual arithmetic conversions are applied consistently to all integer types.

This concept is introduced because the effects of integral promotion and implicit conversions are subtle and sometimes dangerous. Integral promotion is an unavoidable feature of the C++ language,

but **the intention of these rules is that the effect of integral promotion should be neutralized by taking no advantage of the widening that occurs with small integer operands**.

The C++ standard does not explicitly define how small integer types would be balanced to a common type in the absence of integral promotion, although it does establish the principles of value-preservation.

When adding operands of type *int*, the developer is obliged to ensure that the result of the operation will not exceed a value that can be represented in type *int*. If the developer should fail to do so, overflow will occur, and the result will be undefined. It is the intention of the approach described here that the same principle should apply when small integer operands are added; the developer should ensure, for example, that the result of adding two *unsigned chars* can be represented in an *unsigned char*, even though integral promotion could give rise to evaluation in a larger type. In other words, the limitations of the *underlying type* of an expression should be observed, rather than the actual type.

In general, the *underlying type* of an expression is determined by the type of the operand or sub-expression that "is widest". A compliant expression is one in which all operators notionally take operands of this *underlying type*.

Underlying type of an integer literal

The numeric value "5" can be expressed as a literal constant of type *int, unsigned int, long or unsigned long* by the addition (or absence) of a suitable suffix; but no suffix is available to create a representation of the value of *signed char*, *unsigned char* or *short* types. This presents a difficulty when attempting to maintain type consistency in expressions. If it is desired to assign a value to an object of type *unsigned char*, then either an implicit type conversion from an integer type must be tolerated, or else a cast must be introduced. Many would argue that to use a cast in such circumstances serves only to reduce readability.

The same problem exists when literals are required in initializers, function arguments or arithmetic expressions. However, the problem is largely a philosophical one associated with the aspiration to observe the principles of strong typing.

One way of addressing this problem is to **imagine** that an integer literal (or an integer constant expression made up only of literals) has a type appropriate to its magnitude and context. This type then becomes the *underlying type* of the literal.

The *underlying type* of an integer constant expression is therefore defined as follows:

1. If the actual type of the expression is signed integral, the *underlying type* is defined as the smallest signed integer type that is capable of representing its value.
2. If the actual type of the expression is unsigned integral, the *underlying type* is defined as the smallest unsigned integer type that is capable of representing its value.
3. In all other circumstances, the *underlying type* of the expression is defined as being the same as its actual type.

In a 32-bit architecture, the *underlying type* of an integer literal will be determined according to its magnitude and signedness as follows:

Unsigned values

0U	to	255U	8 bit unsigned
256U	to	65535U	16 bit unsigned
65536U	to	4294967295U	32 bit unsigned

Signed values

-2147483648	to	-32769	32 bit signed
-32768	to	-129	16 bit signed
-128	to	127	8 bit signed
128	to	32767	16 bit signed
32768	to	2147483647	32 bit signed

Notice that *underlying type* is an artificial concept. It does not in any way influence the type of evaluation that is actually performed. The concept has been developed simply as a way of defining a safe framework in which to construct arithmetic expressions.

The *cvalue* expression

As a general principle, all operations in an expression should be performed in a consistent type. Therefore, an operation evaluated in one *underlying type* should not be subsequently converted to a different *underlying type*.

An expression that should not undergo further conversions, either implicitly or explicitly, is called a *cvalue* expression. Note that the term *complex expression*, which was used in MISRA C:2004, has the same meaning as *cvalue* in this document.

Determination of the underlying type of an expression

The following describes the mechanism for determining the *underlying type* of an expression.

Class type operands

If any operand has class type and an implicit conversion was performed to convert the class to a built-in type, the *underlying type* shall be the type after the implicit conversion.

Bit-field operands

Bit-field objects have an *underlying type* equivalent to an integral type of a size determined by their width. For example, a bit-field with width *n*, will have the same *underlying type* as a fundamental type with the same sign and width (if one exists).

Underlying type balancing

The following are conceptual replacements for the *usual arithmetic conversions* ([1] §5(9)). These replacements are called the *underlying type conversions*.

- If an overloaded operator function is called, the *underlying type* of the expression is the *underlying type* of the return of the overloaded operator.
- Otherwise, if either operand has *char* type (not explicitly *signed* or *unsigned*), then the *underlying type* of the expression is *char* type.
- Otherwise, if either operand has *enum* type then the *underlying type* of the expression is *enum* type.
- Otherwise, if either operand has pointer type, then the *underlying type* of the expression is pointer type.
- Otherwise, if either operand has *bool* type, then the *underlying type* of the expression is *bool* type.
- Otherwise, if both operands are integral literals (or expressions wholly comprised thereof) then the *underlying type* of the expression is the smallest fundamental type of the appropriate sign required to store the value of the evaluated expression. For example, the *underlying type* of `120+5+3` is `S16`.
- Otherwise, if both operands have integral type, the *underlying type* of the expression can be found using the following:
  - If the types of the operands are the same size, and either is *unsigned*, the result is *unsigned*.
  - Otherwise, the type of the result is that of the larger type.
- Otherwise, if one of the operands has floating point type, then the *underlying type* of the expression can be found using the following:
  - If one of the operands has integral type, then the type of the result is that of the floating point type.
  - Otherwise, the type is the result of the larger type.

The mechanism

Expressions ([1] §5), other than constant integral expressions ([1] §5.19), used in the following contexts are always *cvalues*:

- Function argument expressions;
- Return expressions.

Similarly, unless listed below:

- Binary expressions are not *cvalues* and their *underlying type* is the result of applying the *underlying type conversions*.
- The other unlisted expressions are not *cvalues* and have the *underlying type* of the operation.

Primary expressions ([1] §5.1)

literal

The *underlying type* of an integral literal is the smallest fundamental type of the appropriate sign required to store its value. For example, the *underlying type* of the literal 128 is `S16`. The result is not a *cvalue*.

(expression)

A parenthesized expression is a *cvalue* if the *expression* is a *cvalue*. The *underlying type* is the *underlying type* of the expression.

Postfix expressions ([1] §5.2)

postfix-expression ++
postfix-expression --

The result is a *cvalue* expression whose *underlying type* is that of the *postfix-expression*.

postfix-expression [expression]

The result is not a *cvalue*. The *underlying type* of the result is the *underlying type* of the array element.

postfix-expression (expression-listopt)

The result is not a *cvalue*. The *underlying type* of the result is the *underlying type* of the function return type. For brevity, this pattern is to apply to all function call syntaxes.

simple-type-specifier (expression-listopt)

The result is not a *cvalue*. The *underlying type* of the result is the *underlying type* of *simple-type-specifier*.

dynamic_cast < type-id > (expression)
static_cast < type-id > (expression)
reinterpret_cast < type-id > (expression)
const_cast < type-id > (expression)

The result is not a *cvalue*. The *underlying type* of the result is the *underlying type* of *type-id*.

Unary expressions ([1] §5.3)

++ *cast-expression*
-- *cast-expression*
~ *cast-expression*
- *cast-expression*

The result is a *cvalue* expression whose *underlying type* is that of the *cast-expression*.

! cast-expression

The result is a *cvalue* expression whose *underlying type* is *bool*.

Explicit type conversions ([1] §5.4)

(type-id) cast-expression

The result is not a *cvalue* expression. Its *underlying type* is that of *type-id*.

Multiplicative operators ([1] §5.6)

*multiplicative-expression * pm-expression*
multiplicative-expression / pm-expression
multiplicative-expression % pm-expression

The result is a *cvalue* expression whose *underlying type* is as defined by the *underlying type conversions*.

Additive operators ([1] §5.7)

additive-expression + multiplicative-expression

The result is a *cvalue* expression whose *underlying type* is as defined by the *underlying type conversions*.

additive-expression - multiplicative-expression

The result is a *cvalue* expression. Normally, the *underlying type* of the result is as defined by the *underlying type conversions*; however, where both operands have pointer type, the *underlying type* of the result is *ptrdiff_t*.

Shift operators ([1] §5.8)

shift-expression << additive-expression
shift-expression >> additive-expression

The result is a *cvalue* expression. The *underlying type* of the result is the *underlying type* of the *shift-expression*.

Relational operators ([1] §5.9)

relational-expression < shift-expression
relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression

The result is a *cvalue* expression whose *underlying type* is *bool*.

Equality operators ([1] §5.10)

equality-expression == relational-expression
equality-expression != relational-expression

The result is a *cvalue* expression whose *underlying type* is *bool*.

Bitwise AND operator ([1] §5.11)

and-expression & equality-expression

The result is a *cvalue* expression whose *underlying type* is as defined by the *underlying type conversions*.

Bitwise exclusive OR operator ([1] §5.12)

exclusive-or-expression ^ *and-expression*

The result is a *cvalue* expression whose *underlying type* is as defined by the *underlying type conversions*.

Bitwise inclusive OR operator ([1] §5.13)

inclusive-or-expression | *exclusive-or-expression*

The result is a *cvalue* expression whose *underlying type* is as defined by the *underlying type conversions*.

Logical AND operator ([1] §5.14)

logical-and-expression && *inclusive-or-expression*

This is a *cvalue* expression whose *underlying type* is *bool*.

Logical OR operator ([1] §5.15)

logical-or-expression || *logical-and-expression*

This is a *cvalue* expression whose *underlying type* is *bool*.

Conditional operator ([1] §5.16)

logical-or-expression ? expression : assignment-expression

This is a *cvalue* expression whose *underlying type* is defined by the *underlying type conversions* applied to the *expression* and *assignment-expression*.

Assignment operators ([1] §5.17)

logical-or-expression assignment-operator assignment-expression
assignment-operator: one of
= *= /= %= += -= >>= <<= &= ^= |=

This is a *cvalue* expression whose *underlying type* is the *underlying type* of the *logical-or-expression*.

Comma operator ([1] §5.18)

expression , assignment-expression

This is a *cvalue* expression whose *underlying type* is the *underlying type* of the *assignment-expression*.

Constant expressions ([1] §5.19)

The result is not a *cvalue* expression. The *underlying type* for a constant expression “e” with a value “v” will have the same signedness as “e”, and a magnitude given by the *underlying type* of a single *integer-literal* with the same value as “v”.

Rule 5–0–1	(Required)	The value of an expression shall be the same under any order of evaluation that the standard permits.

[Unspecified 5(4), Undefined 5(4)]

Rationale

Apart from a few operators (notably `&&`, `||`, `?:` and `,` (comma)) the order in which sub-expressions are evaluated is unspecified and can vary. This means that no reliance can be placed on the order of evaluation of sub-expressions and, in particular, no reliance can be placed on the order in which side effects occur. Those points in the evaluation of an expression at which all previous side effects can be guaranteed to have taken place are called "sequence points". Sequence points and side effects are described in Section 1.9(7) of ISO/IEC 14882:2003 [1].

Note that the "order of evaluation" problem is not solved by the use of parentheses, as this is not a precedence issue.

Example

The following notes give some guidance on how dependence on order of evaluation may occur, and therefore may assist in adopting the rule.

- ***increment or decrement operators***

  As an example of what can go wrong, consider

  ```
  x = b[ i ] + i++;
  ```

  This will give different results depending on whether `b[ i ]` is evaluated before `i++` or vice versa. The problem could be avoided by putting the increment operation in a separate statement. For example:

  ```
  x = b[ i ] + i;
  i++;
  ```

- ***function arguments***

  The order of evaluation of function arguments is unspecified.

  ```
  x = func( i++, i );
  ```

  This will give different results depending on which of the function's two parameters is evaluated first.

- ***function pointers***

  If a function is called via a function pointer there shall be no dependence on the order in which function-designator and function arguments are evaluated.

  ```
  p->task_start_fn (p++);
  ```

- ***function calls***

  Functions may have additional effects when they are called (e.g. modifying some global data). Dependence on order of evaluation could be avoided by invoking the function prior to the expression that uses it, making use of a temporary variable for the value.

For example

```
x = f(a) + g(a);
```

could be written as

```
x = f(a);
x += g(a);
```

As an example of what can go wrong, consider an expression to take two values off a stack, subtract the second from the first, and push the result back on the stack:

```
push( pop() - pop() );
```

This will give different results depending on which of the `pop()` function calls is evaluated first (because `pop()` has side effects).

- ***nested assignment statements***

  Assignments nested within expressions cause additional side effects. The best way to avoid any possibility of this leading to a dependence on order of evaluation is not to embed assignments within expressions.

  For example, the following is not recommended:

```
y = 4;
x = y = y++; // It is undefined whether the final value of y
             // is 4 or 5.
```

- ***accessing a volatile***

  The *volatile* type qualifier is provided in C++ to denote objects whose value can change independently of the execution of the program (for example an input register). If an object of volatile qualified type is accessed this may change its value. C++ compilers will not optimize out reads of a volatile. In addition, as far as a C++ program is concerned, a read of a volatile has a side effect (changing the value of the volatile).

  It will usually be necessary to access volatile data as part of an expression, which then means there may be dependence on order of evaluation. Where possible, though, it is recommended that volatiles only be accessed in simple assignment statements, such as the following:

```
volatile uint16_t v;
// ...
x = v;
```

The rule addresses the order of evaluation problem with side effects. Note that there may also be an issue with the number of times a sub-expression is evaluated, which is not covered by this rule. This can be a problem with function invocations where the function is implemented as a macro. For example, consider the following function-like macro and its invocation:

```
#define MAX(a, b) ( ((a) > (b)) ? (a) : (b) )
// ...
z = MAX( i++, j );
```

The definition evaluates the first parameter twice if `a` > `b` but only once if `a` ≤ `b`. The macro invocation may thus increment `i` either once or twice, depending on the values of `i` and `j`.

It should be noted that magnitude-dependent effects, such as those due to floating-point rounding, are also not addressed by this rule. Although the order in which side effects occur is undefined, the result of an operation is otherwise well-defined and is controlled by the structure of the expression. In the following example, `f1` and `f2` are floating-point variables; `F3`, `F4` and `F5` denote expressions with floating-point types.

```
f1 = F3 + ( F4 + F5 );
f2 = ( F3 + F4 ) + F5;
```

The addition operations are, or at least appear to be, performed in the order determined by the position of the parentheses, i.e. firstly `F4` is added to `F5` then secondly `F3` is added to give the value of `f1`. Provided that `F3`, `F4` and `F5` contain no side effects, their values are independent of the order in which they are evaluated. However, the values assigned to `f1` and `f2` are not guaranteed to be the same because floating-point rounding following the addition operations are dependent on the values being added.

Rule 5–0–2	**(Advisory)**	**Limited dependence should be placed on C++ operator precedence rules in expressions.**

Rationale

In addition to the use of parentheses to override default operator precedence, parentheses should also be used to emphasize it. It is easy to make a mistake with the rather complicated precedence rules of C++, and this approach helps to avoid such errors, and helps to make the code easier to read. However, too many parentheses can clutter the code and make it unreadable.

Example

The following guidelines are suggested for deciding when parentheses are required:

- Parentheses are not required for the right-hand operand of an assignment operator unless the right-hand side itself contains an assignment expression:

```
x = a + b;      // acceptable
x = (a + b);    // () not required
```

- Parentheses are not required for the operand of a unary operator:

```
x = a * -1;     // acceptable
x = a * (-1);   // () not required
```

- Otherwise, the operands of binary and ternary operators shall be *cast-expressions* (see Section 5.4(2) of ISO/IEC 14882:2003 [1]) unless all the operators in the expression are the same.

```
x = a + b + c;                  // acceptable, but care needed
x = f ( a + b, c );             // no () required for a + b
x = ( a == b ) ? a : ( a – b );
if ( a && b && c )              // acceptable
x = ( a + b ) – ( c + d );
x = ( a * 3 ) + c + d;
x = static_cast< uint16_t > ( a ) + b; // no need for cast
```

- Even if all operators are the same, parentheses may be used to control the order of operation. Some operators (e.g. addition and multiplication) that are associative in algebra are not necessarily associative in C++. Similarly, integer operations involving mixed types (prohibited by several rules) may produce different results because of the integral promotions. The following example written for a 16-bit implementation demonstrates that addition is not associative and that it is important to be clear about the structure of an expression:

```
uint16_t a = 10U;
uint16_t b = 65535U;
uint32_t c = 0U;
uint32_t d;

d = (a + b) + c;   // d is 9; a + b wraps modulo 65536
d = a + (b + c);   // d is 65545
// this example also deviates from several other rules
```

Note that Rule 5–2–1 is a special case of this rule applicable solely to the logical operators, `&&` and `||`.

Rule 5–0–3	(Required)	A *cvalue* expression shall not be implicitly converted to a different *underlying type*.

Rationale

In order to ensure all operations in an expression are performed in the same *underlying type*, an expression defined as a *cvalue* shall not undergo further implicit conversions.

Example

```
void f ( )
{
   int32_t s32;
   int8_t  s8;

   s32 = s8 + s8;                                  // Example 1 –
                                                   //         Non-compliant
   s32 = static_cast < int32_t > ( s8 ) + s8; // Example 2 - Compliant
   s32 = s32 + s8;                                 // Example 3 - Compliant
}
```

In Example 1, the addition operation is performed with an *underlying type* of *int8_t* and the result is converted to an *underlying type* of *int32_t*.

In Examples 2 and 3, the addition is performed with an *underlying type* of *int32_t* and therefore no *underlying type* conversion is required.

Rule 5–0–4	(Required)	An implicit integral conversion shall not change the signedness of the *underlying type*.

Rationale

Some signed to unsigned conversions may lead to *implementation-defined behaviour*. This behaviour may not be consistent with developer expectations.

Example

```
void f()
{
   int8_t  s8;
   uint8_t u8;
```

```
   s8 = u8;                                    // Non-compliant
   u8 = s8 + u8;                               // Non-compliant
   u8 = static_cast< uint8_t > ( s8 ) + u8;    // Compliant
}
```

Rule 5–0–5	(Required)	There shall be no implicit *floating-integral* conversions.

[Undefined 4.9(1)]

Rationale

Conversions from floating point to integral types discard information, and may lead to *undefined behaviour* if the floating-point value cannot be represented in the integral type.

Conversions from integral types to floating point types may not result in an exact representation, which may not be consistent with developer expectations.

Example

```
void f ( )
{
   float32_t f32;
   int32_t   s32;
   s32 = f32;                                  // Non-compliant
   f32 = s32;                                  // Non-compliant
   f32 = static_cast< float32_t > ( s32 );     // Compliant
}
```

Rule 5–0–6	(Required)	An implicit integral or floating-point conversion shall not reduce the size of the *underlying type*.

[Undefined 4.9(1)]

Rationale

An implicit conversion that results in the size of a type being reduced may result in a loss of information.

Example

```
void f ( )
{
   int32_t s32;
   int16_t s16;
   s16 = s32;                                  // Non-compliant
   s16 = static_cast< int16_t > ( s32 );       // Compliant
}
```

Rule 5–0–7	(Required)	There shall be no explicit *floating-integral* conversions of a *cvalue* expression.

Rationale

A cast applied to the result of an expression does not change the type in which the expression is evaluated, which may be contrary to developer expectations.

Example

```
// Integral to Float
void f1 ( )
{
   int16_t   s16a;
   int16_t   s16b;
   int16_t   s16c;
   float32_t f32a;

   // The following performs integer division
   f32a = static_cast< float32_t > ( s16a / s16b );   // Non-compliant

   // The following also performs integer division
   s16c = s16a / s16b;
   f32a = static_cast< float32_t > ( s16c );          // Compliant

   // The following performs floating-point division
   f32a = static_cast< float32_t > ( s16a ) / s16b;   // Compliant
}
```

In the above example, the expression `( s16a / s16b )` is performed with an *underlying type* of `int16_t` rather than `float32_t`.

```
// Float to Integral
void f2 ( )
{
   float32_t f32a;
   float32_t f32b;
   float32_t f32c;
   int16_t   s16a;

   // The following performs floating-point division
   s16a = static_cast< int16_t > ( f32a / f32b );   // Non-compliant

   // The following also performs floating-point division
   f32c = f32a / f32b;
   s16a = static_cast< int16_t > ( f32c );          // Compliant

   // The following performs integer division
   s16a = static_cast< int16_t > ( f32a ) / f32b;   // Compliant
}
```

In the above example, the expression `( f32a / f32b )` is performed with an *underlying type* of `float32_t` rather than `int16_t`.

Rule 5–0–8	(Required)	An explicit integral or floating-point conversion shall not increase the size of the *underlying type* of a *cvalue* expression.

Rationale

A cast applied to the result of an expression does not change the type in which the expression is evaluated, which may be contrary to developer expectations.

Example

```
void f ( )
{
   int16_t s16;
   int32_t s32;

   s32 = static_cast< int32_t > ( s16 + s16 );    // Non-compliant
   s32 = static_cast< int32_t > ( s16 ) + s16 ;   // Compliant
}
```

In the above example, the expression `( s16 + s16 )` is performed with an *underlying type* of `int16_t` rather than `int32_t`.

Rule 5–0–9	(Required)	An explicit integral conversion shall not change the signedness of the *underlying type* of a *cvalue* expression.

Rationale

A signed to unsigned conversion may lead to an expression having a value inconsistent with developer expectations.

Example

```
void f ( )
{
   int8_t  s8;
   uint8_t u8;

   s8 = static_cast< int8_t >( u8 + u8 );   // Non-compliant
   s8 = static_cast< int8_t >( u8 )
      + static_cast< int8_t >( u8 );        // Compliant
}
```

In the above example, the expression `( u8 + u8 )` is performed with an *underlying type* of `uint8_t` rather than `int8_t`.

Rule 5–0–10	(Required)	If the bitwise operators ~ and << are applied to an operand with an *underlying type* of *unsigned char* or *unsigned short*, the result shall be immediately cast to the *underlying type* of the operand.

Rationale

When the operators ~ and << are applied to *small integer types* (*unsigned char* or *unsigned short*), the operations are preceded by integral promotion, and the result may unexpectedly contain high order bits.

Exception

The immediate assignment of the result obtained by the use of ~ or << on an operand of type *unsigned char* or *unsigned short* to an object of the same *underlying type* complies with this rule (including use as a function argument or function return value), even though the conversion is implicit.

Example

```
uint8_t port = 0x5aU;
uint8_t result_8;
uint16_t result_16;
uint16_t mode;

result_8 = ( ~port ) >> 4;   // Non-compliant
```

`~port` is 0xffa5 on a 16-bit machine but 0xffffffa5 on a 32-bit machine. In either case the value of `result` is 0xfa, but 0x0a may have been expected. This danger is avoided by inclusion of the cast as shown below:

```
result_8  = ( static_cast< uint8_t > (~port) ) >> 4 ;  // Compliant
```

A similar problem exists when the << operator is used on *small integer types* and high order bits are retained. For example:

```
result_16 = ( ( port << 4 ) & mode ) >> 6;             // Non-compliant
```

The value in `result_16` will depend on the implemented size of an *int*. Addition of a cast avoids any ambiguity.

```
result_16 =
   ( static_cast < uint16_t > ( static_cast< uint16_t > ( port ) << 4 )
   & mode ) >> 6;                                      // Compliant
```

Using intermediate steps would make this clearer:

```
uint16_t port_16      = static_cast< uint16_t > ( port );
uint16_t port_shifted = static_cast< uint16_t > ( port_16 << 4 );

result_16 = ( port_shifted & mode ) >> 6;              // Compliant
```

Rule 5–0–11	**(Required)**	**The plain *char* type shall only be used for the storage and use of character values.**

[Implementation 3.9.1(1), 7.1.5.2(1)]

Rationale

The *char* type within C++ is defined for use with the implementation character set. It is *implementation-defined* if *char* is signed or unsigned, and it is therefore unsuitable for use with numeric data.

Character values consist of character literals or strings. A character set maps text characters onto numeric values; the character value is the text itself.

Note that Rule 3–9–2 applies, so this rule also covers the *char_t* type.

Example

```
char_t a = 'a';   // Compliant
char_t b = '\r';  // Compliant
char_t c = 10;    // Non-compliant
char   d = 'd';   // Compliant with this rule, but breaks Rule 3-9-2
```

See also

Rule 3–9–2, Rule 5–0–12

Rule 5–0–12	(Required)	***signed char* and *unsigned char* type shall only be used for the storage and use of numeric values.**

[Implementation 3.9.1(1), 7.1.5.2(1)]

Rationale

There are three distinct *char* types, (plain) *char*, *signed char* and *unsigned char*. *signed char* and *unsigned char* shall only be used for numeric data and plain *char* shall only be used for character data. As it is *implementation-defined*, the signedness of the plain *char* type should not be assumed.

Note that Rule 3–9–2 also applies, so the *uint8_t* and *int8_t* types are covered by this rule.

Example

```
    int8_t  a = 'a';  // Non-compliant – explicitly signed
    uint8_t b = '\r'; // Non-compliant – explicitly unsigned
    int8_t  c = 10;   // Compliant
    uint8_t d = 12U;  // Compliant
signed char e = 11;   // Compliant with this rule, but breaks Rule 3-9-2
```

See also

Rule 3–9–2, Rule 5–0–11

Rule 5–0–13	(Required)	**The *condition* of an *if-statement* and the *condition* of an *iteration-statement* shall have type *bool*.**

Rationale

If an expression with type other than *bool* is used in the *condition* of an *if-statement* or *iteration-statement*, then its result will be implicitly converted to *bool*. The *condition* expression shall contain an explicit test (yielding a result of type *bool*) in order to clarify the intentions of the developer.

Exception

A condition of the form *type-specifier-seq declarator* is not required to have type *bool*.

This exception is introduced because alternative mechanisms for achieving the same effect are cumbersome and error-prone.

Example

```
extern int32_t * fn  ( );
extern int32_t   fn2 ( );
extern bool      fn3 ( );
while ( int32_t * p = fn ( ) )     // Compliant by exception
{
   // Code
}
// The following is a cumbersome but compliant example
do
{
   int32_t * p = fn ( );
   if ( NULL == p )
   {
      break;
   }
   // Code...
}
while ( true );                    // Compliant
while ( int32_t length = fn2 ( ) ) // Compliant by exception
{
   // Code
}
while ( bool flag = fn3 ( ) )      // Compliant
{
   // Code
}
if ( int32_t * p = fn ( ) )        // Compliant by exception
if ( int32_t length = fn2 ( ) )    // Compliant by exception
if ( bool flag = fn3 ( ) )         // Compliant
if ( u8 )                          // Non-compliant
if ( u8 && ( bool_1 <= bool_2 ) )  // Non-compliant
for ( int32_t x = 10; x; --x )     // Non-compliant
```

Rule 5–0–14	**(Required)**	**The first operand of a *conditional-operator* shall have type *bool*.**

Rationale

If an expression with type other than *bool* is used as the first operand of a *conditional-operator*, then its result will be implicitly converted to *bool*. The first operand shall contain an explicit test (yielding a result of type *bool*) in order to clarify the intentions of the developer.

Example

```
int32_a = int16_b ? int32_c : int32_d;          // Non-compliant
int32_a = bool_b  ? int32_c : int32_d;          // Compliant
int32_a = ( int16_b < 5 ) ? int32_c : int32_d;  // Compliant
```

Rule 5–0–15	(Required)	Array indexing shall be the only form of pointer arithmetic.

Rationale

Array indexing is the only acceptable form of pointer arithmetic, because it is clearer and hence less error prone than pointer manipulation. This rule bans the explicit calculation of pointer values. Array indexing shall only be applied to objects defined as an array type.

Any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Pointers may go out of bounds of arrays or structures, or may even point to effectively arbitrary locations.

Exception

The increment/decrement operators may be used on iterators implemented by pointers to an array.

Example

```
template < typename IterType >
uint8_t sum_values ( IterType iter, IterType end )
{
   uint8_t result = 0;

   while ( iter != end )
   {
      result += *iter;
      ++iter;              // Compliant by exception
   }

   return result;
}

void my_fn ( uint8_t * p1, uint8_t p2[ ] )
{
   uint8_t index = 0;
   uint8_t * p3;
   uint8_t * p4;

   *p1 = 0;
   ++index;
   index = index + 5;

   p1      = p1 + 5;     // Non-compliant – pointer increment
   p1[ 5 ] = 0;          // Non-compliant – p1 was not declared as array
   p3      = &p1[ 5 ];   // Non-compliant – p1 was not declared as array

   p2[ 0 ]     = 0;
   p2[ index ] = 0;          // Compliant
   p4          = &p2[ 5 ];   // Compliant
}

uint8_t a1[ 16 ];
uint8_t a2[ 16 ];

my_fn ( a1, a2 );
my_fn ( &a1[ 4 ], &a2[ 4 ] );

uint8_t a[ 10 ];
uint8_t * p;
```

```
p          = a;
*( p + 5 ) = 0;   // Non-compliant
p[ 5 ]     = 0;   // Compliant
sum_values ( &a1[ 0 ], &a1[ 16 ] );
```

See also

Rule 0–3–1, Rule 5–0–16

Rule 5–0–16	(Required)	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

[Undefined 5.7(5)]

Rationale

This rule applies to expressions of the form:

- `integer_expression + pointer_expression`
- `pointer_expression + integer_expression`
- `pointer_expression - integer_expression`
- `++pointer_expression`
- `pointer_expression++`
- `--pointer_expression`
- `pointer_expression--`
- `pointer_expression [ integer_expression ]`

where `pointer_expression` is a pointer to an array element.

It is *undefined behaviour* if the result obtained from one of the above expressions is not a pointer to an element of the array pointed to by `pointer_expression` or an element one beyond the end of that array.

Example

```
void f1 ( const int32_t * a1 )
{
   int32_t a2[ 10 ];
   const int32_t * p1 = &a1 [  1 ];   // Non-compliant – a1 not an array
         int32_t * p2 = &a2 [ 10 ];   // Compliant
         int32_t * p3 = &a2 [ 11 ];   // Non-compliant
}
void f2 ( )
{
   int32_t b;
   int32_t c [ 10 ];
   f1 ( &b );
   f1 ( c );
}
```

See also

Rule 5–0–15

Rule 5–0–17	(Required)	Subtraction between pointers shall only be applied to pointers that address elements of the same array.

[Undefined 5.7(6)]

Rationale

This rule applies to expressions of the form:

```
pointer_expression_1 - pointer_expression_2
```

where `pointer_expression_1` and `pointer_expression_2` are pointers to array elements.

It is *undefined behaviour* if `pointer_expression_1` and `pointer_expression_2` do not point to elements of the same array or the element one beyond the end of that array.

Example

```
void f1 ( )
{
   int32_t a1[ 10 ];
   int32_t a2[ 10 ];

   int32_t * p1 = &a1 [  1 ];
   int32_t * p2 = &a2 [ 10 ];
   int32_t   diff;

   diff = p1 - a1;              // Compliant
   diff = p2 - a2;              // Compliant
   diff = p1 - p2;              // Non-compliant
}
```

Rule 5–0–18	(Required)	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Rationale

Attempting to make comparisons between pointers will produce *undefined behaviour* if the two pointers do not point to the same object.

Note: it is permissible to address the next element beyond the end of an array, but accessing this element is not allowed.

Example

```
void f1 (  )
{
   int32_t a1[ 10 ];
   int32_t a2[ 10 ];
   int32_t * p1 = a1;
   if ( p1 < a1 )  // Compliant
   {
   }
   if ( p1 < a2 )  // Non-compliant
   {
   }
}
```

Rule 5–0–19	(Required)	The declaration of objects shall contain no more than two levels of pointer indirection.

Rationale

Use of more than two levels of indirection can seriously impair the ability to understand the behaviour of the code, and therefore should be avoided.

Example

```
typedef int8_t * INTPTR;
struct s {
   int8_t *   s1;           // Compliant
   int8_t **  s2;           // Compliant
   int8_t *** s3;           // Non-compliant
};
struct s *   ps1;           // Compliant
struct s **  ps2;           // Compliant
struct s *** ps3;           // Non-compliant
int8_t **  (  *pfunc1)();   // Compliant
int8_t **  ( **pfunc2)();   // Compliant
int8_t **  (***pfunc3)();   // Non-compliant
int8_t *** ( **pfunc4)();   // Non-compliant
void function( int8_t *   par1,                // Compliant
               int8_t **  par2,                // Compliant
               int8_t *** par3,                // Non-compliant
               INTPTR *   par4,                // Compliant
               INTPTR *   const * const par5,  // Non-compliant
               int8_t *   par6[],              // Compliant
               int8_t **  par7[])              // Non-compliant
{
```

```
   int8_t *   ptr1;                 // Compliant
   int8_t **  ptr2;                 // Compliant
   int8_t *** ptr3;                 // Non-compliant
   INTPTR *   ptr4;                 // Compliant
   INTPTR *   const * const ptr5;   // Non-compliant
   int8_t *   ptr6[ 10 ];           // Compliant
   int8_t **  ptr7[ 10 ];           // Compliant
}
```

Explanation of types

- `par1` and `ptr1` are of type pointer to `int8_t`.
- `par2` and `ptr2` are of type pointer to pointer to `int8_t`.
- `par3` and `ptr3` are of type pointer to a pointer to a pointer to `int8_t`. This is three levels and is non-compliant.
- `par4` and `ptr4` are expanded to a type of pointer to a pointer to `int8_t`.
- `par5` and `ptr5` are expanded to a type of const pointer to a const pointer to a pointer to `int8_t`. This is three levels and is non-compliant.
- `par6` is of type pointer to pointer to `int8_t` because arrays are converted to a pointer to the initial element of the array.
- `ptr6` is of type pointer to array of `int8_t`.
- `par7` is of type pointer to pointer to pointer to `int8_t` because arrays are converted to a pointer to the initial element of the array. This is three levels and is non-compliant.
- `ptr7` is of type array of pointer to pointer to `int8_t`. This is compliant.

Rule 5–0–20	**(Required)**	**Non-constant operands to a binary bitwise operator shall have the same *underlying type*.**

Rationale

Using operands of the same *underlying type* documents that it is the number of bits in the final (promoted and balanced) type that are used, and not the number of bits in the original types of the expression.

Example

```
uint8_t  mask = ~(0x10);
uint16_t value;

value ^= mask;   // Non-compliant
```

The intent may have been to invert all bits except for bit 5, but the top 8 bits will not have been inverted.

Rule 5–0–21	(Required)	Bitwise operators shall only be applied to operands of unsigned *underlying type*.

[Implementation 5.8(3)]

Rationale

Bitwise operations (`~`, `<<`, `<<=`, `>>`, `>>=`, `&`, `&=`, `^`, `^=`, `|` and `|=`) are not normally meaningful on signed integers or enumeration constants. Additionally, an *implementation-defined* result is obtained if a right shift is applied to a negative value.

Example

```
if ( ( uint16_a & int16_b  ) == 0x1234U )   // Non-compliant
if ( ( uint16_a | uint16_b ) == 0x1234U )   // Compliant
if ( ~int16_a == 0x1234U )                  // Non-compliant
if ( ~uint16_a == 0x1234U )                 // Compliant
```

6.5.2 Postfix expressions

Rule 5–2–1	(Required)	Each operand of a logical `&&` or `||` shall be a *postfix-expression*.

Rationale

The effect of this rule is to require that operands are appropriately parenthesized. Parentheses are important in this situation both for readability of code and for ensuring that the behaviour is as the developer intended.

Exception

Where an expression consists of either a sequence of only logical `&&` or a sequence of only logical `||`, extra parentheses are not required.

Example

```
if ( x == 0 && ishigh )       // Non-compliant
if ( ( x == 0 ) && ishigh )   // Compliant
if ( x || y || z )            // Compliant by exception,
                              //   if x, y and z bool
if ( x || y && z     )        // Non-compliant
if ( x || ( y && z ) )        // Compliant
if ( x && !y )                // Non-compliant
if ( x && ( !y ) )            // Compliant
if ( is_odd( y ) && x )       // Compliant
if ( ( x > c1 ) &&   ( y > c2 ) && ( z > c3 )   )  // Compliant -
                                                   //       exception
if ( ( x > c1 ) &&   ( y > c2 ) || ( z > c3 )   )  // Non-compliant
if ( ( x > c1 ) && ( ( y > c2 ) || ( z > c3 ) ) )  // Compliant as
                                                   //       extra() used
```

Note that this rule is a special case of Rule 5–0–2.

Rule 5–2–2	(Required)	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of *dynamic_cast*.

[Undefined 5.2.9(5, 8)]

Rationale

Casting from a virtual base to a derived class, using any means other than *dynamic_cast* has *undefined behaviour*. The behaviour for *dynamic_cast* is defined.

Example

```
class B  { ... };
class D: public virtual B { ... };

D d;
B *pB  = &d;
D *pD  = static_cast<D*>(pB);   // Non-compliant - undefined behaviour
D *pD2 = dynamic_cast<D*>(pB);  // Compliant, but pD2 may be NULL
D & D3 = dynamic_cast<D&>(*pB); // Compliant, but may throw an exception
```

Rule 5–2–3	(Advisory)	Casts from a base class to a derived class should not be performed on polymorphic types.

Rationale

A downcast occurs when a class type is converted to another class type that is derived from that first class.

Polymorphism enables strong abstraction between the interface and implementation of a hierarchy. Explicit casts bypass this layer of abstraction resulting in higher levels of coupling and dependency.

Example

```
class Colour { /* ... */ };
void setColour ( Colour const & );

class Obj
{
public:
   virtual bool hasColour ( ) const = 0;
   virtual Colour getColour ( ) const = 0;
};
```

```
class ObjWithColour : public Obj
{
public:
   virtual bool hasColour ( ) const
   {
      return true;
   }
   virtual Colour getColour ( ) const
   {
      return m_colour;
   }
private:
   Colour m_colour;
};
void badPrintObject ( Obj const & obj )
{
   ObjWithColour const * pObj =
      dynamic_cast<ObjWithColour const*>( &obj );   // Non-compliant
   if ( 0 != pObj )
   {
      setColour ( pObj->getColour ( ) );
   }
}
void goodPrintObject ( Obj const & obj )
{
   if ( obj.hasColour ( ) )
   {
      setColour ( obj.getColour ( ) );
   }
}
```

The function `badPrintObject` now requires knowledge of how objects in the `Obj` hierarchy are structured. In the future, the hierarchy may be changed so that objects are split into specific colours, and any clients dependent on the colour will then have to be modified to include this change. Clients using virtual functions however, will remain unchanged.

Rule 5–2–4	**(Required)**	**C-style casts (other than *void* casts) and functional notation casts (other than explicit constructor calls) shall not be used.**

Rationale

C-style (*cast* notation), and functional notation casts that do not invoke a converting constructor are capable of performing casts between unrelated types.

Exception

A C-style cast to *void* may be used to signify that the return value for a non-void function call is being ignored (see Rule 0–1–7).

Example

```
class A
{
public:
   explicit A( int32_t );
};
int32_t g ( )
{
   return 7;
}
void f ( )
{
   A const a1 = A( 10 );            // Compliant
   A * a2 = ( A* )( &a1 );          // Non-compliant
   A * a3 = const_cast<A*>( &a1 );  // Compliant, but breaks Rule 5-2-5
   (void)g ( );                     // Compliant by exception
}
```

In the above example, the C-style cast from `a1` to a non-const pointer is stronger than necessary. If the type of `a1` is changed at some future date, then the cast may continue to compile.

See also

ISO/IEC 14882:2003 [1] §5.2.3, §5.4

Rule 5–2–5	**(Required)**	**A cast shall not remove any *const* or *volatile* qualification from the type of a pointer or reference.**

[Undefined 7.1.5.1(4, 7)]

Rationale

Removal of the *const* or *volatile* qualification may not meet developer expectations as it may lead to *undefined behaviour*.

Example

```
void f ( const char_t * p )
{
   *const_cast< char_t * >( p ) = '\0';   // Non-compliant
}
int main ( )
{
   f ( "Hello World!" );
}
```

Rule 5–2–6	(Required)	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

[Undefined 5.2.10(6), Unspecified 5.2.10(6)]

Rationale

Conversion of a function pointer to a non-function pointer type causes *undefined behaviour*.

Undefined behaviour may arise if a function call is made using a pointer that is the result of a function pointer conversion.

Example

```
void f ( int32_t )
{
   reinterpret_cast< void (*)( ) >( &f );   // Non-compliant
   reinterpret_cast< void * >( &f );        // Non-compliant
}
```

Rule 5–2–7	(Required)	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

[Unspecified 5.2.10(7)]

Rationale

The result of converting from a pointer to an unrelated type is unspecified.

Example

```
struct S
{
   int32_t i;
   int32_t j;
};
class C
{
public:
   int32_t i;
public:
   int32_t j;
   virtual ~C ( );
};
void f ( S * s )
{
   C       * c = reinterpret_cast< C * >( s );      // Non-compliant
   int32_t   i = reinterpret_cast< int32_t >( s );  // Compliant, but
                                                    // breaks Rule 5-2-9
   C       * d = reinterpret_cast< C * >( i );      // Non-compliant
   S       * e = reinterpret_cast< S * >( i );      // Compliant, but
                                                    // breaks Rule 5-2-8
}
```

Rule 5–2–8	(Required)	An object with integer type or pointer to *void* type shall not be converted to an object with pointer type.

[Unspecified 5.2.10(7)]

Rationale

In general, converting from an integral type or a pointer to *void* type to a pointer to an object leads to *unspecified behaviour*.

Example

```
struct S
{
   int32_t i;
   int32_t j;
};
void f ( void * v, int32_t i )
{
   S * s1 = reinterpret_cast< S * >( v );   // Non-compliant
   S * s2 = reinterpret_cast< S * >( i );   // Non-compliant
}
```

Rule 5–2–9	(Advisory)	A cast should not convert a pointer type to an integral type.

[Implementation 5.2.10(4, 5)]

Rationale

The size of integer that is required when a pointer is converted to an integer is *implementation-defined*. Casting between a pointer and an integer type should be avoided where possible, but may be unavoidable when addressing memory mapped registers or other hardware specific features.

Note that C++ does not permit a pointer to be converted to any floating type.

Example

```
struct S
{
   int32_t i;
   int32_t j;
};
void f ( S * s )
{
   int32_t p = reinterpret_cast< int32_t >( s );   // Non-compliant
}
```

Rule 5–2–10	(Advisory)	The increment (++) and decrement (- -) operators should not be mixed with other operators in an expression.

Rationale

The use of increment and decrement operators in combination with other arithmetic operators is not recommended, because:

- It can significantly impair the readability of the code.
- It introduces additional side effects into a statement, with the potential for *undefined behaviour*.

It is safer to use these operators in isolation from any other arithmetic operators.

Example

A statement such as the following is non-compliant:

```
u8a = ++u8b + u8c--;   // Non-compliant
```

The following sequence is clearer and therefore safer:

```
++u8b;
u8a = u8b + u8c;
u8c--;
```

See also

ISO/IEC 14882:2003 [1] §5.2.6, §5.3.2

Rule 5–2–11	(Required)	The comma operator, && operator and the || operator shall not be overloaded.

Rationale

Overloaded versions of the comma and logical conjunction operators have the semantics of function calls whose sequence point and ordering semantics are different from those of the built-in versions. It may not be clear at the point of use that these operators are overloaded, and so developers may be unaware which semantics apply.

Example

```
#include "util.h"

class A
{
public:
   UtilType getValue ( );
   UtilType setValue ( UtilType const & );
};

void f1 ( A & a1, A & a2 )
{
   a1.getValue ( ) && a2.setValue ( 0 );   // Short circuiting may occur
}
```

```
bool operator && ( UtilType const &,
                   UtilType const & );     // Non-compliant
void f2 ( A & a1, A & a2 )
{
   a1.getValue ( ) && a2.setValue ( 0 );   // Both operands evaluated
}
```

If the type returned by `getValue` and `setValue` has an overloaded operator `&&`, then both `getValue` and `setValue` will be evaluated.

See also

ISO/IEC 14882:2003 [1] §5.14, §5.15, §5.18

Rule 5–2–12	(Required)	An identifier with array type passed as a function argument shall not decay to a pointer.

Rationale

When a variable with array type decays to a pointer, its bounds are lost.

If a design requires arrays of different lengths, then a class should be used to encapsulate the array objects and so ensure that the dimensionality is maintained.

Example

```
void f1( int32_t p[ 10 ] );
void f2( int32_t *p );
void f3( int32_t ( &p )[ 10 ] );

void b ()
{
   int32_t a[ 10 ];
   f1( a );      // Non-compliant - Dimension "10" lost due to array to
                 // pointer conversion.
   f2( a );      // Non-compliant - Dimension "10" lost due to array to
                 // pointer conversion.
   f3( a );      // Compliant - Dimension preserved.
}
```

6.5.3 Unary expressions

Rule 5–3–1	(Required)	Each operand of the `!` operator, the logical `&&` or the logical `||` operators shall have type *bool*.

Rationale

The use of operands with types other than *bool* with these operators is unlikely to be meaningful (or intended). This rule allows the detection of such uses, which often occur because the logical operators (`&&`, `||` and `!`) can be easily confused with the bitwise operators (`&`, `|` and `~`).

Example

```
if ( ( a < b ) && ( c < d ) )   // Compliant
if ( 1 && ( c < d ) )           // Non-compliant
if ( ( a < b ) && ( c + d ) )   // Non-compliant

if ( u8_a && ( c + d ) )        // Non-compliant
if ( !0 )                       // Non-compliant –
                                //   also breaks other rules
if ( !ptr )                     // Non-compliant
if ( !false )                   // Compliant with this rule,
                                // but breaks others
```

See also

ISO/IEC 14882:2003 [1] §5.14, §5.15

Rule 5–3–2	(Required)	The unary minus operator shall not be applied to an expression whose *underlying type* is unsigned.

Rationale

Applying the unary minus operator to an expression of type *unsigned int*, *unsigned long* or *unsigned long long* generates a result of type *unsigned int*, *unsigned long* or *unsigned long long* respectively and is not a meaningful operation. Applying unary minus to an operand of smaller unsigned integer type may generate a meaningful signed result due to integral promotion, but this is not considered good practice.

Example

On a machine with a 32-bit *int* type:

```
uint8_t  a = -1U;   // Non-compliant – a is assigned 255
int32_t  b = -a;    // Non-compliant – b is assigned -255
uint32_t c = 1U;
int64_t  d = -c;    // Non-compliant – d is assigned MAX_UINT
```

See also

See Section 6.5.0 for a description of *underlying type*.

Rule 5–3–3	(Required)	The unary & operator shall not be overloaded.

[Undefined 5.3.1(4)]

Rationale

Taking the address of an object of incomplete type where the complete type contains a user declared operator & leads to *undefined behaviour*.

Example

```
// A.h
class A
{
public:
   A * operator & ( );   // Non-compliant
};
```

```
// f1.cc
class A;
void f ( A & a )
{
   &a;   // uses built-in operator &
}
// f2.cc
#include "A.h"
void f2 ( A & a )
{
   &a;   // use user-defined operator &
}
```

See also

ISO/IEC 14882:2003 [1] §5.3.1(4)

Rule 5–3–4	(Required)	Evaluation of the operand to the *sizeof* operator shall not contain side effects.

Rationale

A possible programming error in C++ is to apply the *sizeof* operator to an expression and expect the expression to be evaluated. However, the expression is not evaluated as *sizeof* only acts on the type of the expression. To avoid this error, *sizeof* shall not be used on expressions that would contain side effects if they were used elsewhere, as the side effects will not occur.

Exception

An operand of the form `sizeof ( i )` where `i` is volatile is permitted.

Example

```
          int32_t i;
          int32_t j;
volatile int32_t k;
j = sizeof( i = 1234 );   // Non-compliant - j is set to the sizeof the
                          // type of i which is an int32_t.
                          // i is not set to 1234.
j = sizeof ( k );         // Compliant by exception.
```

See also

ISO/IEC 14882:2003 [1] §3.2(2)

6.5.8 Shift operators

Rule 5–8–1	**(Required)**	**The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the *underlying type* of the left hand operand.**

[Undefined 5.8(1)]

Rationale

It is *undefined behaviour* if the right hand operand is negative, or greater than or equal to the width of the left hand operand.

If, for example, the left hand operand of a left-shift or right-shift is a 16-bit integer, then it is important to ensure that this is shifted only by a number between 0 and 15 inclusive.

There are various ways of ensuring that this rule is followed. The simplest is for the right hand operand to be a constant (whose value can then be statically checked). Use of an unsigned integer type will ensure that the operand is non-negative, so then only the upper limit needs to be checked (dynamically at run time or by review). Otherwise both limits will need to be checked.

Example

```
u8a  = (uint8_t) ( u8a << 7 );              // Compliant
u8a  = (uint8_t) ( u8a << 9 );              // Non-compliant
u16a = (uint16_t)( (uint16_t) u8a << 9 );   // Compliant
```

See also

See Section 6.5.0 for a description of *underlying type*.

6.5.14 Logical AND operator

Rule 5–14–1	**(Required)**	**The right hand operand of a logical `&&` or `||` operator shall not contain side effects.**

Rationale

There are some situations in C++ where certain parts of expressions may not be evaluated. If these sub-expressions contain side effects then those side effects may or may not occur, depending on the values of other sub expressions.

The operators which can lead to this problem are `&&` and `||` where the evaluation of the right-hand operand is conditional on the value of the left-hand operand. The conditional evaluation of the right-hand operand of one of the logical operators can easily cause problems if the developer relies on a side effect occurring.

Example

```
if ( ishigh && ( x == i++ ) )      // Non-compliant
...
if ( ishigh && ( x == f( x ) ) )   // Only acceptable if f(x) is
                                   // known to have no side effects
```

The operations that cause side effects are accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations, which cause changes in the state of the execution environment of the calling function.

See also

Rule 5–2–11

ISO/IEC 14882:2003 [1] §3.2(2), §5.14, §5.15

6.5.17 Assignment operators

Rule 5–17–1	**(Required)**	**The semantic equivalence between a binary operator and its assignment operator form shall be preserved.**

Rationale

Where a set of operators is overloaded, it is important that the interactions between the operators meet developer expectations.

Example

```
class A
{
public:
   A& operator= ( A const & rhs );
};

A & operator += ( A const & lhs, A const & rhs );
A const operator + ( A const & lhs, A const & rhs );

void f ( A a1, A a2 )
{
   A x;
   x = a1 + a2;          // Example 1
   a1 += a2;             // Example 2

   if ( x == a1 )        // Example 3
   {
   }
}
```

For a built-in type, the results of Example 1 and Example 2 will be the same, therefore the condition in Example 3 should always be true. This should also be true for overloaded versions of these operators.

6.5.18 Comma operator

Rule 5–18–1	**(Required)**	**The comma operator shall not be used.**

Rationale

Use of the comma operator is generally detrimental to the readability of code, and the same effect can be achieved by other means.

Example

```
f ( ( 1, 2 ), 3 );   // Non-compliant – how many parameters?
```

6.5.19 Constant expressions

Rule 5–19–1	(Advisory)	Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Rationale

Unsigned integer expressions do not strictly overflow, but instead wrap around in a modular way. Any constant unsigned integer expressions that in effect "overflow" will not be detected by the compiler. Although there may be good reasons at run-time to rely on the modular arithmetic provided by unsigned integer types, the reasons for using it at compile-time to evaluate a constant expression are less obvious. Any instance of an unsigned integer constant expression wrapping around is therefore likely to indicate a programming error.

This rule applies equally to all phases of the translation process. Constant expressions that the compiler chooses to evaluate at compile time are evaluated in such a way that the results are identical to those that would be obtained by evaluation on the target, with the exception of those appearing in conditional preprocessing directives. For such directives, the usual rules of arithmetic apply **but** the *int* and *unsigned int* types behave instead as if they were *long* and *unsigned long* respectively.

Example

On a machine with a 16-bit *int* type and a preprocessor using a 32-bit *long* type:

```
#define START   0x8000
#define END     0xFFFF
#define LEN     0x8000

#if ( ( START + LEN ) > END )
#error Buffer Overrun // OK as START and LEN are unsigned long
#endif

#if ( ( ( END – START ) – LEN ) < 0 )
   #error Buffer Overrun
   // Not OK: subtraction result wraps around to 0xFFFFFFFF
#endif

// contrast the above START + LEN with the following
void fn ( )
{
   if ( ( START + LEN ) > END )
   {
      error ( "Buffer overrun" );
      // Not OK: START + LEN wraps around to 0x0000 due to unsigned int
      // arithmetic
   }
}
```

6.6 Statements

6.6.2 Expression statement

Rule 6–2–1	(Required)	Assignment operators shall not be used in sub-expressions.

Rationale

Assignments used in a sub-expression add an additional side effect to that of the full expression, potentially resulting in a value inconsistent with developer expectations. In addition, this helps to avoid getting = and == confused.

Example

```
x = y;
x = y = z;                  // Non-compliant
if ( x != 0 )               // Compliant
{
   foo ( );
}
bool b1 = x != y;           // Compliant
bool b2;
b2 = x != y;                // Compliant
if ( ( x = y ) != 0 )       // Non-compliant
{
   foo ( );
}
if ( x = y )                // Non-compliant
{
   foo ( );
}
if ( int16_t i = foo ( ) )  // Compliant
{
}
```

Rule 6–2–2	(Required)	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Rationale

The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true, even when they are expected to. Also, the behaviour of such a comparison cannot be predicted before execution, and may well vary from one implementation to another.

The recommended method for achieving deterministic floating-point comparisons is to write a library that implements the comparison operations. The library should take into account the floating-point granularity (*std::numeric_limits<float>::epsilon()*) and the magnitude of the numbers being compared.

Example

The result of the test in the following code is unpredictable:

```
float32_t x, y;
if ( x == y )      // Non-compliant
if ( x == 0.0f )   // Non-compliant
```

An indirect test is equally problematic and is also prohibited by this rule:

```
if ( ( x <= y ) && ( x >= y ) )   // Non-compliant
if ( ( x < y ) || ( x > y ) )     // Non-compliant
```

The following is better, but only if the magnitudes are appropriate:

```
if ( fabs ( x – y ) <=
     std::numeric_limits<float>::epsilon( ) )   // Compliant
```

Rule 6–2–3	(Required)	**Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.**

Rationale

Null statements should not normally be included deliberately, but where they are used, they shall appear on a line by themselves. White-space characters may precede the null statement to preserve indentation. If a comment follows the null statement, then at least one white-space character shall separate the null statement from the comment. The use of a white-space character to separate the null statement from any following comment is required on the grounds that it provides an important visual cue to reviewers. Following this rule enables a static checking tool to warn of null statements appearing on a line with other text, which would normally indicate a programming error.

Example

```
while ( ( port & 0x80 ) == 0 )
{
    ; // wait for pin - Compliant
    /* wait for pin */ ; // Non-compliant, comment before ;
    ;// wait for pin – Non-compliant, no white-space char after ;
}
```

6.6.3 Compound statement

Rule 6–3–1	(Required)	**The statement forming the body of a *switch*, *while*, *do ... while* or *for* statement shall be a compound statement.**

Rationale

If the bodies of these statements are not compound statements, then errors can occur if a developer fails to add the required braces when attempting to change a single statement body to a multi-statement body.

Requiring that the body of a *switch* statement or a *while*, *do ... while* or *for* loop shall be a compound statement (enclosed within braces) ensures that these errors cannot arise.

Example

```
for ( i = 0; i < N_ELEMENTS; ++i )
{                                       // Compliant
   buffer [ i ] = 0;                    // Even a single statement must
                                        // be in braces
}
for ( i = 0; i < N_ELEMENTS; ++i );     // Non-compliant
                                        // Accidental single null statement
{
   buffer [ i ] = 0;
}
while ( new_data_available ) // Non-compliant
   process_data ( );         // Incorrectly not enclosed in braces
   service_watchdog ( );     // Added later but, despite the appearance
                             // (from the indent) it is actually not
                             // part of the body of the while statement,
                             // and is executed only after the loop has
                             // terminated
```

Note that this example assumes a particular style for the layout of compound statements and their enclosing braces. This style is not mandated, but a style should be defined within the style guide for the project.

6.6.4 Selection statements

Rule 6–4–1	**(Required)**	**An *if (condition)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement.**

Rationale

If the bodies of these constructs are not compound statements, then errors can occur if a developer fails to add the required braces when attempting to change a single statement body to a multi-statement body.

Requiring that the body of these constructs shall be a compound statement (enclosed within braces) ensures that these errors cannot arise.

Example

```
if ( test1 );      // Non-compliant - accidental single null statement
{
   x = 1;
}
if ( test1 )
{
   x = 1;          // Compliant - a single statement must be in braces
}
else if ( test2 )  // Compliant - no need for braces between else and if
{
   x = 0;          // Compliant – a single statement must be in braces
}
```

```
else                // Non-compliant
   x = 3;           // This was (incorrectly) not enclosed in braces
   y = 2;           // This line was added later but, despite the
                    // appearance (from the indent) it is actually not
                    // part of the else, and is executed unconditionally
```

Note that this example assumes a particular style for the layout of compound statements and their enclosing braces. This style is not mandated, but a style should be defined within the style guide for the project.

Rule 6–4–2	**(Required)**	**All *if … else if* constructs shall be terminated with an *else* clause.**

Rationale

When an *if* statement is followed by one or more *else if* statements then the final *else if* shall be followed by an *else* statement. In the case of a simple *if* statement the *else* statement need not be included.

The final *else* statement, which should either take appropriate action or contain a suitable comment as to why no action is taken, is defensive programming.

Example

For example this code is a simple *if* statement:

```
if ( x < 0 )
{
   log_error( 3 );
   x = 0;
}
// else not needed
```

Whereas the following code demonstrates an *if, else if* construct

```
if ( x < 0 )
{
   log_error ( 3 );
   x = 0;
}
else if ( y < 0 )
{
   x = 3;
}
else      // this else clause is required, even if the
{         // developer expects this will never be reached
   // No change in value of x
}
```

Rule 6–4–3	**(Required)**	**A *switch* statement shall be a *well-formed switch statement*.**

A *well-formed switch statement* conforms to the following syntax rules, which are additional to the C++ standard syntax rules. All syntax rules not defined below are as defined in ISO/IEC 14882:2003 [1].

switch-statement:
`switch (` *condition* `) {` *case-label-clause-list default-label-clause*$_{opt}$ `}`

case-label-clause-list:
case-label case-clause$_{opt}$
case-label-clause-list case-label case-clause$_{opt}$

case-label:
`case` *constant-expression* `:`

case-clause:
case-block-seq$_{opt}$ `break ;`
case-block-seq$_{opt}$ `throw` *assignment-expression*$_{opt}$ `;`
`{` *statement-seq*$_{opt}$ `break ; }`
`{` *statement-seq*$_{opt}$ `throw` *assignment-expression*$_{opt}$ `; }`

default-label-clause:
default-label default-clause

default-label:
`default :`

default-clause:
case-clause

case-block:
expression_statement
compound_statement
selection_statement
iteration_statement
try_block

case-block-seq:
case-block
case-block-seq case-block

The following statements, which are permitted by C++, are explicitly not included within the MISRA C++ *switch* syntax rules. Note, however, that they are permitted within the compound statements forming the body of a *switch-clause*.

labelled_statement
jump_statement
declaration_statement

The following terms are also used within the text of the rules:

switch-label	Either a *case-label* or *default-label*.
case-clause	The code between any two *switch-labels*.
default-clause	The code between the *default-label* and the end of the *switch* statement.
switch-clause	Either a *case-clause* or a *default-clause*.

Rationale

The syntax for the *switch* statement in C++ is weak, allowing complex, unstructured behaviour. The previous text describes the syntax for *switch* statements as defined by MISRA C++. This, and the associated rules, imposes a simple and consistent structure on to the *switch* statement.

Example

```
switch ( x )
{
case 0:
   ...
   break;           // break is required here
case 1:             // empty clause, break not required
case 2:
   break;           // break is required here
default:            // default clause is required
   break;           // break is required here, in case a future
                    // modification turns this into a case clause
}
```

Rule 6–4–4	(Required)	A *switch-label* shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement.

Rationale

A *switch-label* can be placed anywhere within the statements that form the body of a *switch* statement, potentially leading to unstructured code. To prevent this from happening, the scope of a *case-label* or *default-label* shall be the compound statement forming the body of a *switch* statement. All *case-clauses* and the *default-clause* shall be at the same scope.

Example

```
switch ( x )
{
case 1:             // Compliant
   if ( ... )
   {
   case 2:          // Non-compliant
      DoIt ( );
   }
   break;
default:
   break;
}
```

Rule 6–4–5	(Required)	An unconditional *throw* or *break* statement shall terminate every non-empty *switch-clause*.

Rationale

If a developer fails to add a *break* statement to the end of a *switch-clause*, then control flow "falls" into any following *switch-clause*. Whilst this is sometimes intentional, it is often an error.

To ensure that such errors can be detected, the last statement in every *switch-clause* shall be a *break* statement, or if the *switch-clause* is a compound statement, then the last statement in the compound statement shall be a *break* statement.

A special case exists if the *switch-clause* is empty, as this allows groups of clauses requiring identical statements to be created.

Example

```
switch ( x )
{
case 0:
   break;           // Compliant
case 1:             // Compliant - empty drop through
case 2:             //             allows a group
   break;           // Compliant
case 3:
   throw;           // Compliant
case 4:
   a = b;
                    // Non-compliant - non empty drop through
default:
   ;                // Non-compliant – default must also have "break"
}
```

Rule 6–4–6	(Required)	The final clause of a *switch* statement shall be the *default-clause*.

Rationale

The requirement for a final *default-clause* is defensive programming. This clause shall either take appropriate action, or else contain a suitable comment as to why no action is taken.

Exception

If the *condition* of a *switch* statement is of type *enum*, and all the enumerators are listed in case labels, then the *default-clause* is not required as the rules associated with *enums* are intended to ensure that the *enum* cannot be assigned values outside of its set of enumerators. Note that it may still be appropriate to include a *default-clause* for the purpose of defensive programming.

Example

```
switch ( int16 )
{
case 0:
   break;
case 1:
case 2:
   break;
                // Non-compliant – default clause is required.
 }
enum Colours { RED, BLUE, GREEN } colour;
switch ( colour )
{
   case RED:
      break;
   case GREEN:
      break;
                // Non-compliant – default clause is required.
}
```

```
switch ( colour )
{
   case RED:
      break;
   case BLUE:
      break;
   case GREEN:
      break;
                  // Compliant – exception allows no default in this case
}
```

Rule 6–4–7	(Required)	The *condition* of a *switch* statement shall not have *bool* type.

Rationale

An *if* statement gives a clearer representation for a Boolean choice.

Example

```
switch ( x == 0 )   // Non-compliant
{
   ...
}
```

Rule 6–4–8	(Required)	Every *switch* statement shall have at least one *case-clause*.

Rationale

A *switch* statement with no *case-clauses* is redundant.

Example

```
switch ( x )
{
               // Non-compliant
default:
   break;
}
```

6.6.5 Iteration statements

The *for* statement

ISO/IEC 14882:2003 [1] §6.5.3 states:

The *for* statement

`for (` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

is equivalent to:

```
{
    for-init-statement
    while ( condition_opt )
    {
        statement
        expression_opt ;
    }
}
```

So, it can be seen that the *for* loop in C++ is in effect an alternative syntax for a *while* loop. Because of this, it is possible that an unbounded loop can be created in error. The following rules are designed to reduce the risk of this occurring.

The rules make use of the following definitions:

- A *loop-control-variable* is any variable occurring in *for-init-statement*, *condition* or *expression*.
- A *loop-counter* is a *loop-control-variable* that is:
  (a) Initialized in, or prior to, *for-init-statement*; and
  (b) an operand to a relational operator in *condition*; and
  (c) modified in *expression*.

Note that iterators are also valid *loop-counters*. As an iterator may be a class type, any operator referenced in the following rules may be an overloaded operator.

Rule 6–5–1	**(Required)**	**A *for* loop shall contain a single *loop-counter* which shall not have floating type.**

Rationale

A *for* loop without exactly one *loop-counter* is simply a *while* loop. If this is the desired behaviour, then a *while* loop is more appropriate.

Example

```
y = 0;
for ( x = 0; x < y; x = y++ )   // Non-compliant
```

Rule 6–5–2	(Required)	If *loop-counter* is not modified by `--` or `++`, then, within *condition*, the *loop-counter* shall only be used as an operand to `<=`, `<`, `>` or `>=`.

Rationale

When the *loop-counter* is modified using an operator other than `--` or `++`, then `==` and `!=` shall not be used, as loop termination may not occur, which may be inconsistent with developer expectations.

Example

```
for ( i = 1; i != 10; i += 2 )   // Non-compliant
for ( i = 1; i <= 10; i += 2 )   // Compliant
for ( i = 1; i != 10; ++i )      // Compliant
```

Rule 6–5–3	(Required)	The *loop-counter* shall not be modified within *condition* or *statement*.

Rationale

Modification of the *loop-counter* other than in *expression* leads to a badly-formed *for* loop.

Example

```
bool modify ( int32_t * pX )
{
   *pX++;
   return ( *pX < 10 );
}
for ( x = 0; modify ( &x ); )  // Non-compliant
{
}
for ( x = 0; x < 10; )
{
   x = x * 2;                  // Non-compliant
}
```

Rule 6–5–4	(Required)	The *loop-counter* shall be modified by one of: `--`, `++`, `-=n`, or `+=n`; where `n` remains constant for the duration of the loop.

Rationale

This helps to ensure deterministic loop termination. The other *for* loop rules mean that the modification can only take place within *expression*.

Example

```
for ( x = 0; x < 10; ++x )           // Compliant
for ( T x = thing.start( );
      x != thing.end( );
      ++x )                          // Compliant
for ( x = 0; x < 10; x += 1 )        // Compliant
for ( x = 0; x < 10; x += n )        // Compliant if n is not modified
                                     // within the body of the loop.
for ( x = 0; x < 10; x += fn ( ) )   // Non-compliant
```

Rule 6–5–5 (Required) **A *loop-control-variable* other than the *loop-counter* shall not be modified within *condition* or *expression*.**

Rationale

loop-control-variables are either the *loop-counter*, or flags used for early loop termination. The code is easier to understand if these are not modified within *condition* or *expression*.

Note that it is possible for a *loop-control-variable* with *volatile* qualification to change value (or have it changed) outside *statement* due to the volatile nature of the object. Such modification does not break this rule.

Example

```
for ( x = 0; ( x < 10 ) && !bool_a; ++x )
{
   if ( ... )
   {
      bool_a = true;                          // Compliant
   }
}
bool test_a ( bool * pB )
{
   *pB = ... ? true : false;
   return *pB;
}
for ( x = 0;
      ( x < 10 ) && test_a ( &bool_a );
      ++x )                                   // Non-compliant
volatile bool status;
for ( x = 0; ( x < 10 ) && status; ++x)       // Compliant
for ( x = 0; x < 10; bool_a = test( ++x ) )   // Non-compliant
```

Rule 6–5–6	(Required)	A *loop-control-variable* other than the *loop-counter* which is modified in *statement* shall have type *bool*.

Rationale

loop-control-variables are typically used to terminate a *for* loop early. The code is easier to understand if this is done with the use of Boolean values (flags).

Example

```
for ( x = 0; ( x < 10 ) && ( u8a != 3U ); ++x )      // Non-compliant
{
   uint8_a = fn ( );
}
for ( x = 0; ( x < 10 ) && flag; ++x )               // Compliant
{
   u8a  = fn ( );
   flag = u8a != 3U;
}
```

6.6.6 Jump statements

Rule 6–6–1	(Required)	Any label referenced by a *goto* statement shall be declared in the same block, or in a block enclosing the *goto* statement.

Rationale

Unconstrained use of *goto* can lead to programs that are extremely difficult to comprehend, analyse and, for C++, can also lead to the program exhibiting *unspecified behaviour*.

However, in many cases a total ban on *goto* requires the introduction of flags to ensure correct control flow, and it is possible that these flags may themselves be less transparent than the *goto* they replace.

Therefore, the restricted use of *goto* is allowed where that use will not lead to semantics contrary to developer expectations. Jumping in to nested blocks is prohibited as it may lead to complex flow graphs.

Example

```
void f1 ( )
{
   int32_t j = 0;
   goto L1;
   for ( j = 0; j < 10 ; ++j )
   {
L1:                                  // Non-compliant
      j;
   }
}
```

```
void f2 ( )
{
   for ( int32_t j = 0; j < 10 ; ++j )
   {
      for ( int32_t i = 0; i < 10; ++i )
      {
         goto L1;
      }
   }
L1:                                  // Compliant
   f1 ( );
}
```

Rule 6–6–2	(Required)	The *goto* statement shall jump to a label declared later in the same function body.

Rationale

Unconstrained use of *goto* can lead to programs that are extremely difficult to comprehend, analyse and, for C++, can also lead to the program exhibiting *unspecified behaviour*.

However, in many cases a total ban on *goto* requires the introduction of flags to ensure correct control flow, and it is possible that these flags may themselves be less transparent than the *goto* they replace.

Therefore, the restricted use of *goto* is allowed where that use will not lead to semantics contrary to developer expectations. “Back” jumps are prohibited as they can be used to create iterations without using the well-defined iteration statements supplied by the core language.

Example

```
void f ( )
{
   int32_t j = 0;
L1:
   ++j;
   if ( 10 == j )
   {
      goto L2;       // Compliant
   }
   goto L1;          // Non-compliant
L2:
   ++j;
}
```

Rule 6–6–3	(Required)	The *continue* statement shall only be used within a *well-formed for loop*.

Rationale

Over-use of the *continue* statement can lead to unnecessary complexity within the code. This complexity may impede effective testing as extra logic must be tested. The required testing may not be achievable due to control flow dependencies.

A *well-formed for loop* is one which satisfies Rule 6–5–1 to Rule 6–5–6.

Example

```
void fn ( )
{
   for ( int32_t i = 0 ; i != 10; ++i )
   {
      if ( ( i % 2 ) == 0 )
      {
         continue;           // Compliant
      }
      // ...
   }
   int32_t j = -1;
   for ( int32_t i = 0 ; i != 10 && j != i; ++i )
   {
      if ( ( i % 2 ) == 0 )
      {
         continue;           // Non-compliant – loop is not well-formed
      }
      // ...
      ++j;
   }
}
```

Rule 6–6–4	(Required)	For any iteration statement there shall be no more than one *break* or *goto* statement used for loop termination.

Rationale

Restricting the number of exits from a loop is done in the interests of good structured programming. One *break* or *goto* statement is acceptable in a loop since this allows, for example, for dual outcome loops or optimal coding.

Example

```
for ( int32_t i = 0; i < 10; i++ )
{
   if ( ... )
   {
      break;   // Compliant
   }
}
while ( ... )
{
   if ( ... )
   {
      break;   // Compliant
   }
}
```

```
for ( int32_t i = 0; i < 10; i++ )
{
   if ( ... )
   {
      break;
   }
   else if ( ... )
   {
      break;   // Non-compliant – second jump from loop
   }
   else
   {
      ...
   }
}
while ( ... )
{
   if ( ... )
   {
      break;
   }
   if ( ... )
   {
      break;   // Non-compliant – second jump from loop
   }
}
```

Rule 6–6–5	(Required)	A function shall have a single point of exit at the end of the function.

Rationale

This is required by IEC 61508 [12], as part of the requirements for a modular approach.

Exception

A function implementing a *function-try-block* is permitted to have multiple points of exit, one for the try block and one for each catch handler.

Throwing an exception that is not caught within the function is not considered a point of exit for this rule.

Example

```
void fn ( void )
{
   if ( ... )
   {
      return;             // Non-compliant
   }
```

```
   try
   {
      if ( ... )
      {
         throw ( 1 );     // Compliant by exception
      }
   }
   catch ( int32_t )
   {
      throw;              // Compliant by exception
   }
   return;                // Non-compliant
}
void fn2 ( void )
{
   try
   {
      return;             // Non-compliant
   }
   catch ( ... )
   {
      return;             // Non-compliant
   }
}
void fn3 ( void ) try
{
   return;                // Compliant by exception
}
catch ( int32_t )
{
   return;                // Compliant by exception
}
catch ( ... )
{
   return;                // Compliant by exception
}
```

See also

IEC 61508 [12] Part 3 Table B.9

6.7 Declarations

6.7.1 Specifiers

Rule 7–1–1	(Required)	A variable which is not modified shall be *const* qualified.

Rationale

If a variable does not need to be modified, then it shall be declared with *const* qualification so that it cannot be modified. A non-parametric variable will then require its initialization at the point of declaration. Also, future maintenance cannot accidentally modify the value.

Example

```
void b ( int32_t * );
int32_t f (  int32_t * p1,          // Non-compliant
             int32_t * const p2,    // Compliant
             int32_t * const p3 )   // Compliant
{
   *p1 = 10;
   *p2 = 10;
   b( p3 );
   int32_t i = 0;                   // Non-compliant
   return i;
}
```

See also

ISO/IEC 14882:2003 [1] §7.1.5.1

Rule 7–1–2	(Required)	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Rationale

This rule leads to greater precision in the definition of the function interface. The *const* qualification shall be applied to the object pointed to, not to the pointer, since it is the object itself that is being protected.

Exception

This rule does not apply if the parameter object is modified by any of the functions in a set of overriding functions.

Example

```
void myfunc(       int16_t *       param1,
             const int16_t *       param2,
                   int16_t *       param3,
                   int16_t * const param4)
// param1: Addresses an object which is modified     - Compliant
// param2: Addresses an object which is not modified – Compliant
// param3: Addresses an object which is not modified – Non-compliant
// param4: Addresses an object which is not modified – Non-compliant
{
   *param1 = *param2 + *param3 + *param4;
   // Data at address param3 and param4 have not been changed
}
```

6.7.2 Enumeration declarations

Rule 7–2–1	(Required)	An expression with *enum underlying type* shall only have values corresponding to the enumerators of the enumeration.

[Unspecified 7.2(9)]

Rationale

It is *unspecified behaviour* if the evaluation of an expression with *enum underlying type* yields a value which does not correspond to one of the enumerators of the enumeration.

Additionally, other rules in this standard assume that objects of *enum* type only contain values corresponding to the enumerators. This rule ensures the validity of these assumptions.

One way of ensuring compliance when converting to an enumeration is to use a *switch* statement.

Example

```
enum1 convert ( int16_t v )
{
   enum1 result = enum1_ERROR;
   switch ( v )
   {
      case 0:  { result = enum1_E1;    break; }
      case 1:  { result = enum1_E2;    break; }
      default: { throw ( ENUM_ERROR ); break; }
   }
   return result;
}
```

6.7.3 Namespaces

Rule 7–3–1	(Required)	The global namespace shall only contain *main*, namespace declarations and *extern "C"* declarations.

Rationale

Declaring names into appropriate namespaces reduces the names found during lookup, helping to ensure that the names found meet developer expectations.

Exception

The types defined for compliance with Rule 3–9–2 may also be in the global namespace.

Example

```
void f1 ( int32_t );          // Non-compliant
int32_t x1;                   // Non-compliant
namespace
{
   void f2 ( int32_t );       // Compliant
   int32_t x2;                // Compliant
}
```

```
namespace MY_API
{
   void b2 ( int32_t );      // Compliant
   int32_t x2;               // Compliant
}
int32_t main ( )             // Compliant
{
}
```

Rule 7–3–2	(Required)	The identifier *main* shall not be used for a function other than the global function *main*.

[Implementation 3.6.1(2, 3)]

Rationale

main (or its equivalent) is usually the entry point to the program and is the only identifier which must be in the global namespace. The use of *main* for other functions may not meet developer expectations.

Example

```
int32_t main ( )       // Compliant
{
}
namespace
{
   int32_t main ( )    // Non-compliant
   {
   }
}
namespace NS
{
   int32_t main ( )    // Non-compliant
   {
   }
}
```

Rule 7–3–3	(Required)	There shall be no unnamed namespaces in *header files*.

Rationale

An unnamed namespace will be unique within each translation unit. Any declarations appearing in an unnamed namespace in a header will refer to different entities in each translation unit, which may not be consistent with developer expectations.

Example

```
// Header.hpp
namespace                    // Non-compliant
{
   extern int32_t x;
}
```

```
// File1.cpp
#include "Header.hpp"

namespace
{
   int32_t x;
}
void fn_a ( void )
{
   x = 24;
}
// File2.cpp
#include "Header.hpp"

namespace
{
   int32_t x;
}
void fn_b ( void )
{
   fn_a ( );
   if ( x == 24 )            // This x will not have been initialized.
   {
   }
}
```

Rule 7–3–4 (Required) *using-directives* shall not be used.

Rationale

using-directives add additional scopes to the set of scopes searched during name lookup. All identifiers in these scopes become visible, increasing the possibility that the identifier found by the compiler does not meet developer expectations.

using-declarations or fully qualified names restricts the set of names considered to only the name explicitly specified, and so are safer options.

Example

```
namespace NS1
{
   int32_t i1;
   int32_t j1;
   int32_t k1;
}
using namespace NS1;   // Non-compliant

namespace NS2
{
   int32_t i2;
   int32_t j2;
   int32_t k2;
}
using NS2::j2;         // Compliant
```

```
namespace NS3
{
   int32_t i3;
   int32_t j3;
   int32_t k3;
}
void f ()
{
   ++i1;
   ++j2;
   ++NS3::k3;
}
```

In the above, `i1` is found via the *using-directive*. However, as a result of the *using-directive*, `j1` and `k1` are also visible. The *using-declaration* allows `j2` to be found while `i2` and `k2` remain hidden. Finally, the qualified name `NS3::k3` unambiguously refers to `k3`, and `i3`, `j3` and `k3` remain hidden to normal lookup.

See also

ISO/IEC 14882:2003 [1] §7.3.4

Rule 7–3–5	(Required)	Multiple declarations for an identifier in the same namespace shall not straddle a *using-declaration* for that identifier.

Rationale

The set of identifiers introduced by a *using-declaration* does not include any declarations that may be added by a subsequent declaration in the namespace. Any subsequent declarations will not be found through the *using-declaration*, which may not be consistent with developer expectations.

Example

```
namespace NS1
{
   void f( uint16_t );   // Example 1
}
using NS1::f;
namespace NS1
{
   void f( uint32_t );   // Example 2 – Non-compliant
}
void bar()
{
   f( 0U );
}
```

In the above example, moving the *using-declaration* below the second namespace would result in Example 2 being called, as it is a better match than Example 1.

See also

ISO/IEC 14882:2003 [1] §7.3.3

Rule 7–3–6	(Required)	***using-directives*** **and** ***using-declarations*** **(excluding class scope or function scope** ***using-declarations***) **shall not be used in** ***header files***.

Rationale

It is important to ensure that the order of inclusion of *header files* cannot affect the behaviour of a program.

Example

```
// f1.h
void foo ( char_t a );

namespace NS1
{
  void foo( int32_t a );
}

inline void bar ( )
{
   foo ( 0 );
}

// f2.h
namespace NS1
{
}

using namespace NS1;

// f1.cc
#include "f1.h"
#include "f2.h"

int32_t m1 ( )
{
  bar ( );   // bar calls foo ( char_t );
}

// f2.cc
#include "f2.h"
#include "f1.h"

void m2 ( )
{
  bar ( );   // bar calls foo ( int32_t );
}
```

In the above example, changing the order of the *header files* alters the meaning of the program.

See also

ISO/IEC 14882:2003 [1] §7.3.3, §7.3.4

6.7.4 The *asm* declaration

Rule 7–4–1	(Document)	All usage of assembler shall be documented.

[Implementation 7.4(1)]

Rationale

Assembly language code is *implementation-defined* and therefore is not portable.

Rule 7–4–2	(Required)	Assembler instructions shall only be introduced using the *asm* declaration.

Rationale

The *asm* declaration is available to all C++ implementations, allowing a consistent mechanism to be used.

However, the parameters to *asm* are still *implementation-defined*.

Example

```
void Delay_a ( void )
{
   asm ( "NOP" );   // Compliant
}
void Delay_b ( void )
{
#pragma asm
   "NOP"            // Non-compliant
#pragma endasm
}
```

Rule 7–4–3	(Required)	Assembly language shall be encapsulated and isolated.

Rationale

Ensuring that assembly language code is encapsulated and isolated aids portability.

Where assembly language instructions are needed, they shall be encapsulated and isolated in either assembler functions or C++ functions.

Example

```
void Delay ( void )
{
   asm ( "NOP" );    // Compliant
}
void fn ( void )
{
   DoSomething ( );
   Delay ( );        // Assembler is encapsulated
   DoSomething ( );
   asm ( "NOP" );    // Non-compliant
   DoSomething ( );
}
```

6.7.5 Linkage specifications

Rule 7–5–1	**(Required)**	**A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.**

Rationale

Automatic variables are destroyed at the end of the function call. Returning a reference or pointer to such a variable allows it to be used after its destruction, leading to *undefined behaviour*.

Example

```
int32_t * fn1 ( void )
{
   int32_t x = 99;
   return ( &x );              // Non-compliant
}
int32_t * fn2 ( int32_t y )
{
   return ( &y );              // Non-compliant
}
int32_t & fn3 ( void )
{
   int32_t x = 99;
   return ( x );               // Non-compliant
}
int32_t & fn4 ( int32_t y )
{
   return ( y );               // Non-compliant
}
int32_t * fn5 ( void )
{
   static int32_t x = 0;
   return &x;                  // Compliant
}
```

Rule 7–5–2	**(Required)**	**The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.**

Rationale

If the address of an automatic object is assigned to another automatic object of larger scope, or to a static object, or returned from a function, then the object containing the address may exist beyond the time when the original object ceases to exist (and its address becomes invalid).

Note that throwing a pointer to an object with automatic storage is also a violation of this rule.

Example

```
void foobar ( void )
{
   int8_t * p1;
   {
      int8_t local_auto;
      p1 = &local_auto;      // Non-compliant
   }
}
```

Rule 7–5–3	(Required)	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Rationale

It is *implementation-defined behaviour* whether the reference parameter is a temporary object or a reference to the parameter. If the implementation uses a local copy (temporary object), this will be destroyed when the function returns. Any attempt to use such an object after its destruction will lead to *undefined behaviour*.

Example

```
int32_t * fn1 ( int32_t & x )
{
   return ( &x );                    // Non-compliant
}
int32_t * fn2 ( )
{
   int32_t i = 0;
   return fn1 ( i );
}
const int32_t * fn3 ( const int32_t & x )
{
   return ( &x );                    // Non-compliant
}
int32_t & fn4 ( int32_t & x )
{
   return ( x );                     // Non-compliant
}
const int32_t & fn5 ( const int32_t & x )
{
   return ( x );                     // Non-compliant
}
```

Rule 7–5–4	(Advisory)	Functions should not call themselves, either directly or indirectly.

Rationale

Unbounded recursion is likely to lead to a stack over-flow and may impact system timings. This is also the case for an iterative algorithm.

Example

```
int32_t fn ( int32_t x )
{
   if ( x > 0 )
   {
      x = x * fn ( x - 1 );      // Non-compliant
   }
   return ( x );
}
// File1.cpp
int32_t fn_2 ( int32_t x )
{
   if ( x > 0 )
   {
      x = x * fn_3 ( x - 1 );    // Non-compliant
   }
   return ( x );
}
// File2.cpp
int32_t fn_3 ( int32_t x )
{
   if ( x == 0 )
   {
      x = x * fn_2 ( x - 1 );    // Non-compliant
   }
   return ( x );
}
```

6.8 Declarators

6.8.0 General

Rule 8–0–1	(Required)	An *init-declarator-list* or a *member-declarator-list* shall consist of a single *init-declarator* or *member-declarator* respectively.

Rationale

Where multiple declarators appear in the same declaration the type of an identifier may not meet developer expectations.

Example

```
int32_t i1; int32_t j1;   // Compliant
int32_t i2, *j2;          // Non-compliant
int32_t *i3,
        &j3 = i2;         // Non-compliant
```

See also

ISO/IEC 14882:2003 [1] §9.2

6.8.3 Meaning of declarators

Rule 8–3–1	**(Required)**	**Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.**

Rationale

Default arguments are determined by the static type of the object. If a default argument is different for a parameter in an overriding function, the value used in the call will be different when calls are made via the base or derived object, which may be contrary to developer expectations.

Example

```
class Base
{
public:
   virtual void g1 ( int32_t a = 0 );
   virtual void g2 ( int32_t a = 0 );
   virtual void b1 ( int32_t a = 0 );
};
class Derived : public Base
{
public:
   virtual void g1 ( int32_t a = 0 );   // Compliant - same default used
   virtual void g2 ( int32_t a );       // Compliant -
                                        //          no default specified
   virtual void b1 ( int32_t a = 10 );  // Non-compliant - different
value
};
void f( Derived& d )
{
   Base& b = d;
   b.g1 ( );         // Will use default of 0
   d.g1 ( );         // Will use default of 0
   b.g2 ( );         // Will use default of 0
   d.g2 ( 0 );       // No default value available to use
   b.b1 ( );         // Will use default of 0
   d.b1 ( );         // Will use default of 10
}
```

The default argument for `g2` can only be used via the base class object and so the value used will always be the same.

See also

ISO/IEC 14882:2003 [1] §8.3.6(10)

6.8.4 Function definitions

Rule 8–4–1	(Required)	Functions shall not be defined using the ellipsis notation.

[Undefined 5.2.2(7), 18.7(3)]

Rationale

Passing arguments via an ellipsis bypasses the type checking performed by the compiler. Additionally, passing an argument with non-*POD* class type leads to *undefined behaviour*.

Note that the rule specifies "defined" (and not "declared") so as to permit the use of existing library functions.

Example

```
void MyPrintf ( char_t * pFormat, ... );   // Non-compliant
```

Rule 8–4–2	(Required)	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Rationale

The name given to a parameter helps document the purpose of the parameter within the function body. If a function parameter is renamed in a subsequent re-declaration, then having different names for the same object will probably lead to developer confusion.

Note that the rule also applies to any overriding set.

Exception

It is not a violation of this rule if the declaration or re-declaration contains an unnamed parameter.

Example

```
// File1
void CreateRectangle ( uint32_t Height, uint32_t Width );

// File2
// Non-compliant
void CreateRectangle ( uint32_t Width, uint32_t Height );

void fn1 ( int32_t a );
void fn2 ( int32_t   );

void fn1 ( int32_t b )   // Non-compliant
{
}
```

```
void fn2 ( int32_t b )   // Compliant
{
}
```

Rule 8–4–3	(Required)	All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression.

[Undefined 6.6.3(2)]

Rationale

This expression gives the value that the function returns. The absence of a *return* with an expression leads to *undefined behaviour* (and the compiler may not give an error).

Exception

This rule does not apply if a function exit is due to exception handling (i.e. a *throw* statement).

Example

```
int32_t fn1 ( void )
{
}                           // Non-compliant
int32_t fn3 ( int32_t x )
{
   if ( x > 100 )
   {
      throw 42;             // Compliant by exception
   }
   return ( x );            // Compliant
}
```

Rule 8–4–4	(Required)	A function identifier shall either be used to call the function or it shall be preceded by `&`.

Rationale

A function identifier can implicitly convert to a pointer to a function. In certain contexts this may result in a well-formed program, but which is contrary to developer expectations. For example, if the developer writes:

```
if ( f )
```

then it is not clear whether the intent is to test if the address of the function is *NULL* or if a call to the function `f()` should be made and the brackets have been unintentionally omitted. The use of the `&` (*address-of*) operator will resolve this ambiguity.

Exception

Passing the function by reference, or assigning it to a reference object is not a violation of this rule.

Example

```
extern void f ( void );
```

```
if ( 0 == f )            // Non-compliant
{
   // ...
}
void (*p)( void ) = f;   // Non-compliant
if ( 0 == &f )           // Compliant
{
   (f)();                // Compliant as function is called
}
void (*p)( void ) = &f;  // Compliant
```

6.8.5 Initializers

Rule 8–5–1	(Required)	All variables shall have a defined value before they are used.

[Indeterminate 8.5(9)]

Rationale

The intent of this rule is that all variables shall have been written to before they are read. This does not necessarily require initialization at declaration.

Note that according to ISO/IEC 14882:2003 [1], variables with static storage duration are automatically initialized to zero by default, unless explicitly initialized. In practice, many embedded environments do not implement this behaviour. Static storage duration is a property of all variables declared with the *static* storage class specifier, or with external linkage. Variables with automatic storage duration are not usually automatically initialized.

Each class constructor shall initialize all non-static members of its class.

Example

```
class C
{
   public:
      C ( ) : m_a( 10 ), m_b( 7 )   // Compliant
      {
      }
      C ( int32_t a ) : m_a( a )    // Non-compliant
      {
      }
      int32_t GetmB ( void )
      {
         return ( m_b );
      }
   private:
      int32_t m_a;
      int32_t m_b;
};
C c( 5 );
```

```
int main ( void )
{
   if ( c.GetmB( ) > 0 )         // m_b has not been initialized
   {
   }
}
```

Rule 8–5–2	(Required)	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Rationale

ISO/IEC 14882:2003 [1] requires initializer lists for arrays, structures and union types to be enclosed in a single pair of braces (though the behaviour if this is not done is undefined). The rule given here goes further in requiring the use of additional braces to indicate nested structures. This forces the developer to explicitly consider and demonstrate the order in which elements of complex data types are initialized (e.g. multi-dimensional arrays).

The zero initialization of arrays or structures shall only be applied at the top level.

The non-zero initialization of arrays or structures requires an explicit initializer for each element.

Example

The following shows two valid ways of initializing the elements of a two dimensional array, but the first does not adhere to the rule:

```
int16_t y[3][2] = { 1, 2, 3, 4, 5, 6 };             // Non-compliant
int16_t y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } }; // Compliant
```

A similar principle applies to structures, and nested combinations of structures, arrays and other types.

Note also that all the elements of arrays or structures can be initialized (to zero or *NULL*) by giving an explicit initializer for the first element only. If this method of initialization is chosen then the first element should be initialized to zero (or *NULL*), and nested braces need not be used.

```
// The following are compliant
int16_t a1[5]    = { 1, 2, 3, 0, 0 };   // Non-zero initialization
int16_t a2[5]    = { 0 };               // Zero initialization
int16_t a3[2][2] = { };                 // Zero initialization

// The following are non-compliant
int16_t a4[5]    = { 1, 2, 3 };         // Partial initialization
int16_t a5[2][2] = { { }, { 1, 2 } };   // Zero initialization
                                        //        at sub-level
```

Rule 8–5–3	(Required)	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Rationale

If an enumerator list is given with no explicit initialization of members, then C++ allocates a sequence of integers starting at zero for the first element and increasing by one for each subsequent element.

An explicit initialization of the first element, as permitted by the above rule, forces the allocation of integers to start at the given value. When adopting this approach it is essential to ensure that the initialization value used is small enough that no subsequent value in the list will exceed the *int* storage used by enumeration constants.

Explicit initialization of all items in the list, which is also permissible, prevents the mixing of automatic and manual allocation, which is error prone. However it is then the responsibility of the developer to ensure that all values are in the required range, and that values are not unintentionally duplicated.

Example

The following example assigns the same value to the `green` and `yellow` enumeration constants. It is unclear to a reviewer if this was intentional or an error.

```
enum colour { red=3, blue, green, yellow=5 };       // Non-compliant
```

However, if all the items are explicitly initialized, then the duplicated values are acceptable as the duplication is readily detectable by anyone reviewing the code.

```
enum colour { red=3, blue=4, green=5, yellow=5 };   // Compliant
```

6.9 Classes

6.9.3 Member functions

Rule 9–3–1	**(Required)**	**const member functions shall not return non-const pointers or references to *class-data*.**

Rationale

When an object is declared with *const* class type, only const member functions can be invoked on that object. The common expectation of const member functions is that the state of the object may not be modified when invoking the functions. However, returning a non-const pointer or reference to *class-data* from a const function allows a modification to the conceptual state of an object.

Example

```
class C
{
public:
   C ( int32_t & b_ ) : a ( new int32_t [ 10 ] ), b ( b_ )
   {
   }
   int32_t * getA () const          // Non-compliant
                                    // Returns non const pointer to data
   {
      return a;
   }
   int32_t * getB () const          // Non-compliant
                                    // Returns non const pointer to data
   {
      return &b;
   }
```

```
   const int32_t * getC () const   // Compliant
                                   // Returns const pointer to data
   {
      return &b;
   }
private:
   int32_t * a;
   int32_t & b;
};
void fn ( C const & c )
{
   c.getA()[ 0 ] = 0;  // Modification to conceptual state of C
   *c.getB()     = 0;  // Modification to conceptual state of C
   fn2 ( c.getC() );   // Value can be used,
   *c.getC()     = 0;  // but compiler will report an error here
}
```

Rule 9–3–2	(Required)	Member functions shall not return non-const *handles* to *class-data*.

Rationale

By implementing class interfaces with member functions the implementation retains more control over how the object state can be modified and helps to allow a class to be maintained without affecting clients. Returning a *handle* to *class-data* allows for clients to modify the state of the object without using any interfaces.

Example

```
class C
{
public:
   int32_t & getA ()  // Non-compliant
   {
      return a;
   }
private:
   int32_t a;
};
void b ( C & c )
{
   int32_t & a_ref = c.getA ();
   a_ref = 10;                  // External modification of private C::a
}
```

`c.getA()` returns a reference to the member, which is then stored and modified by `a_ref`. The class, therefore, has no control over changes to its state.

Where a *resource* is used by the class, but is not *class-data*, non-const *handles* to this data may be returned.

```
class C
{
public:
   C ( int32_t * shared ) : m_shared ( shared )
   {
   }
   int32_t * getA ()
   {
      return m_shared;  // Compliant -  m_shared is not class-data
   }
private:
   int32_t * m_shared;
};
```

Rule 9–3–3	(Required)	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Rationale

Declaring a member function static or const limits its access to the non-static data members. This helps to prevent unintentional modification of the data, and facilitates compliance with Rule 7–1–1.

Example

```
class A
{
public:
   int16_t f1 ( )   // Non-compliant – can be const
   {
      return m_i;
   }

   int16_t f2 ( )   // Non-compliant – can be static
   {
      return m_s;
   }
   int16_t f3 ( )   // Compliant – cannot be const or static
   {
      return ++m_i;
   }
private:
   int16_t m_i;
   static int16_t m_s;
};
```

6.9.5 Unions

Rule 9–5–1	(Required)	Unions shall not be used.

[Implementation 3.9(4, 5)]

Rationale

The use of unions to access an object in different ways may result in the data being misinterpreted. Therefore, this rule prohibits the use of unions for any purpose.

It is recognized nonetheless that there are situations in which the careful use of unions is desirable in constructing an efficient implementation. In such situations, deviations to this rule are considered acceptable provided that all relevant *implementation-defined behaviour* is documented. This might be achieved in practice by referencing the implementation section of the compiler manuals from the design documentation.

Example

```
namespace NS1
{
                  // Compliant - no union
}
namespace NS2
{
   union U1     // Non-compliant – union
   {
      int32_t   i;
      float32_t j;
   };
}
```

6.9.6 Bit-fields

Rule 9–6–1	(Document)	When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.

[Implementation 9.6(1)]

Rationale

Certain aspects of bit-fields are *implementation-defined*. In particular, the developer should be aware of the following:

- It is *implementation-defined* whether the bit-fields are allocated from the high or low end of a storage unit (usually a byte).
- It is *implementation-defined* whether or not a bit field can overlap a storage unit boundary (e.g. if a 6-bit bit-field and a 4-bit bit-field are declared in that order, then the 4-bit bit-field may either start a new byte or it may use 2 bits in one byte and 2 bits in the next).

These issues are generally benign (e.g. when packing together short-length data to save storage space), but they may lead to errors if the absolute position of the bit-fields is important (e.g. when accessing hardware registers).

Provided the elements of the structure are only accessed by name, the developer need make no assumptions about the way that the bit fields are stored within the structure.

Note that Rule 3–9–2 need not be followed when defining bit-fields, as their lengths are explicitly specified.

If the compiler has a switch to force bit fields to follow a particular layout, then this option should be documented.

Example

```
struct message                  // Struct is for bit-fields only
{
   signed   int  little: 4;     // Note: use of basic types is required
   unsigned int  x_set:  1;
   unsigned int  y_set:  1;
} message_chunk;
```

Rule 9–6–2	**(Required)**	**Bit-fields shall be either *bool* type or an explicitly *unsigned* or *signed* integral type.**

[Implementation 3.9.1(5), 7.1.5.2(1), 9.6(3)]

Rationale

Using *int* is *implementation-defined* because bit-fields of type *int* can be either *signed* or *unsigned*.

The use of *wchar_t* as a bit-field type is prohibited as ISO/IEC 14882:2003 [1] does not explicitly define the underlying representation as signed or unsigned.

Example

```
struct S
{
     signed int   a : 2;   // Compliant
   unsigned int   b : 2;   // Compliant
            char  c : 2;   // Non-compliant
     signed char  d : 2;   // Compliant
   unsigned char  e : 2;   // Compliant
            short f : 2;   // Non-compliant
     signed short g : 2;   // Compliant
   unsigned short h : 2;   // Compliant
            int   i : 2;   // Non-compliant
            bool  j : 2;   // Compliant
         wchar_t  k : 2;   // Non-compliant
        uint32_t  l : 2;   // Compliant
          int8_t  m : 2;   // Compliant
};
```

Rule 9–6–3	(Required)	Bit-fields shall not have *enum* type.

[Undefined DR 58]

Rationale

The use of *enum* as a bit-field type is prohibited as ISO/IEC 14882:2003 [1] does not explicitly define the underlying representation as *signed* or *unsigned*. It is therefore not possible to determine the exact number of bits required to represent all values in the enumeration.

Example

```
struct S
{
   AnEnumType n : 2;   // Non-compliant
};
```

Rule 9–6–4	(Required)	Named bit-fields with *signed* integer type shall have a length of more than one bit.

Rationale

The values which may be represented by a bit-field of length one may not meet developer expectations. Anonymous signed bit-fields of any length are allowed.

Example

```
struct S
{
   signed int   a : 1;   // Non-compliant
   signed int     : 1;   // Compliant
   signed int     : 0;   // Compliant
   signed int   b : 2;   // Compliant
   signed int     : 2;   // Compliant
};
```

6.10 Derived classes

6.10.1 Multiple base classes

Rule 10–1–1	(Advisory)	Classes should not be derived from virtual bases.

Rationale

The use of virtual base classes can introduce a number of undefined and potentially confusing behaviours. The use of virtual bases is not recommended.

Example

```
class B {};
class D: public virtual B {}; // Non-compliant - B is a virtual base
```

See also

Rule 5–2–2, Rule 10–1–2, Rule 12–1–2

Rule 10–1–2	(Required)	A base class shall only be declared virtual if it is used in a diamond hierarchy.

Rationale

The use of virtual base classes can introduce a number of undefined and potentially confusing behaviours. Therefore, a base class shall only be declared virtual if that base class is to be used as a common base class in a diamond hierarchy.

Example

```
class A {};
class B1: public virtual A {};  // Compliant – A is a common base for C
class B2: public virtual A {};  // Compliant – A is a common base for C
class C:  public B1, B2 {};
class D: public virtual A {};   // Non-compliant
```

Rule 10–1–3	(Required)	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Rationale

If a base class is both virtual and non-virtual in a multiple inheritance hierarchy then there will be at least two copies of the base class sub-object in the derived object. This may not be consistent with developer expectations.

Example

```
class A {};
class B1: public virtual A {};
class B2: public virtual A {};
class B3: public         A {};
class C:  public B1, B2, B3 {}; // Non-compliant –
                                //    C has two A sub-objects
```

6.10.2 Member name lookup

Rule 10–2–1	(Advisory)	All accessible entity names within a multiple inheritance hierarchy should be *unique*.

Rationale

If the names are ambiguous, the compiler should report the name clash and not generate arbitrary or unexpectedly resolved code. However, this ambiguity may not be obvious to a developer.

There is also a specific concern that if the member function is virtual, resolving the ambiguity by explicitly referencing the base class in effect removes the virtual behaviour from the function.

Exception

For the purposes of this rule, visible function identifiers that form an overload set shall be considered as the same entity.

Example

```
class B1
{
public:
   int32_t count;   // Non-compliant
   void foo ( );    // Non-compliant
};
class B2
{
public:
   int32_t count;   // Non-compliant
   void foo ( );    // Non-compliant
};
class D : public B1, public B2
{
public:
   void Bar ( )
   {
      ++count;      // Is that B1::count or B2::count?
      foo ( );      // Is that B1::foo() or B2::foo()?
   }
};
```

In the above example, in a member function of `D`, the use of `count` or `foo` is ambiguous and must be disambiguated by `B1::count`, `B2::foo`, etc.

6.10.3 Virtual functions

Rule 10–3–1	(Required)	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

Rationale

The main aim of this rule is clarity for maintainers and reviewers, by ensuring that the version of a function that can be executed from any point in a class hierarchy is unambiguous.

Additionally, where classes form a diamond hierarchy, *call by dominance* ([1] §10.3(11)) may occur resulting in a call to a function that is inconsistent with developer expectations. This rule also prevents *call by dominance*.

Exception

Destructors may be declared virtual in multiple members of a class hierarchy.

If a function is declared pure and defined in the same class, then that definition is ignored for this rule.

Example

```
class A
{
public:
   virtual void f1 ( ) = 0;   // f1 is pure
   virtual void f2 ( ) = 0;   // f2 is pure
   virtual void f3 ( ) { }    // f3 is not pure
   virtual void f4 ( ) = 0;   // f4 is pure
   virtual ~A();              // destructor
};
// A::f1 is both pure and has a definition
void A::f1 ( )
{
}
// A::f4 is both pure and has a definition
void A::f4 ( )
{
}
class B : public A
{
public:
   virtual void f2 ( ) { }    // Compliant: f2 pure in A and
                              //            defined in B
   virtual void f3 ( ) { }    // Non-compliant: f3 defined in A and B
   virtual void f4 ( ) = 0;   // Compliant: f4 is pure in A and in B
   virtual ~B();              // Compliant: destructor
};
// Compliant by Exception - f4 defined in A but also declared pure in A
void B::f4 ( )
{
}
class C : public B
{
public:
   virtual void f1 ( ) { } // Compliant: f1 defined in A and C
                           //            but was pure in A
   virtual void f2 ( ) { } // Non-compliant f2: defined in B and C
                           // and not declared pure in B
   virtual void f4 ( ) { } // Compliant by Exception: f4 defined in A
                           // and B but also declared pure in A and B
};
class D : public C
{
public:
   virtual void f1 ( ) { } // Non-compliant f1: defined in C and D
                           // as well as A
   virtual ~D();           // Compliant: destructor
};
```

```
// Call by dominance example
class V
{
public:
   virtual void foo ( )
   {
   }
};
class B1 : public virtual V
{
public:
   virtual void foo ( )  // Non-compliant
   {
   }
};
class B2 : public virtual V
{
public:
   void f1 ( )
   {
      foo();  // V::foo would appear to be the only
              // candidate to be called here
   }
};
class D : public B1, public B2
{
public:
   void f2 ( )
   {
      f1();
   }
};
B2 b2;
b2.f1();  // calls V::foo by normal inheritance rules
D d;
d.f2();   // calls B2::f1 which now calls B1::foo
          // "by dominance"
d.f1();   // also calls B1::foo "by dominance"
```

Rule 10–3–2	**(Required)**	**Each overriding virtual function shall be declared with the *virtual* keyword.**

Rationale

Declaring overriding virtual functions with the *virtual* keyword removes the need to check the base class to determine whether a function is virtual.

Example

```
class A
{
public:
   virtual void g();
   virtual void b();
};
class B1 : public A
{
public:
   virtual void g();  // Compliant     - explicitly declared "virtual"
   void b();          // Non-compliant - implicitly virtual
};
```

Rule 10–3–3	**(Required)**	**A virtual function shall only be overridden by a *pure virtual function* if it is itself declared as *pure virtual*.**

Rationale

Re-declaring a function as pure may not meet developer expectations.

Example

```
class A
{
public:
   virtual void foo ( ) = 0;   // foo declared pure
};
class B : public A
{
public:
   virtual void foo ( )        // foo defined
   {
   }
};
class C: public B
{
public:
   virtual void foo ( ) = 0;   // Non-compliant – foo re-declared pure
};
```

The function `foo` is introduced as pure (making class `A` abstract), defined in class `B` (making class `B` concrete), then re-declared as pure (making class `C` abstract). As this may not meet developer expectations, the re-declaration as pure is not allowed.

6.11 Member access control

6.11.0 General

Rule 11–0–1	(Required)	Member data in non-*POD* class types shall be private.

Rationale

By implementing class interfaces with member functions, the implementation retains more control over how the object state can be modified, and helps to allow a class to be maintained without affecting clients.

Example

```
class C
{
public:
   int32_t b;      // Non-compliant
protected:
   int32_t c;      // Non-compliant
private:
   int32_t d;      // Compliant
};
```

6.12 Special member functions

6.12.1 Constructors

Rule 12–1–1	(Required)	An object's dynamic type shall not be used from the body of its constructor or destructor.

[Undefined 10.4(6)]

Rationale

During construction and destruction of an object, its final type may be different to that of the completely constructed object. The result of using an object's dynamic type in a constructor or destructor may not be consistent with developer expectations.

The dynamic type of an object is used in the following constructs:

- *typeid* on a class with a virtual function or a virtual function in a base class.
- *dynamic_cast*
- A virtual call to a virtual function.

This rule also prohibits a call being made to a *pure virtual* function from within a constructor or destructor. Such calls lead to *undefined behaviour*.

Example

```
class B1
{
public:
   B1 ( )
   {
      typeid ( B1 );                  // Compliant, B1 not polymorphic
   }
};
class B2
{
public:
   virtual ~B2 ( );
   virtual void foo ( );
   B2 ( )
   {
      typeid ( B2 );                  // Non-compliant
      B2::foo ( );                    // Compliant – not a virtual call
      foo ( );                        // Non-compliant
      dynamic_cast< B2* > ( this );   // Non-compliant
   }
};
```

Rule 12–1–2	(Advisory)	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

Rationale

This rule reduces confusion over which constructor will be used, and with what parameters.

Example

```
class A
{
public:
   A ( )
   {
   }
};
class B : public A
{
public:
   B ( )   // Non-compliant – benign, but should be B ( ) : A ( )
   {
   }
};
```

```
class V
{
public:
   V ( )
   {
   }
   V ( int32_t i )
   {
   }
};
class C1 : public virtual V
{
public:
   C1 ( ) : V ( 21 )
  {
  }
};
class C2 : public virtual V
{
public:
   C2 ( ) : V ( 42 )
   {
   }
};
class D: public C1, public C2
{
public:
   D ( )   // Non-compliant
   {
   }
};
```

There would appear to be an ambiguity here, as `D` only includes one copy of `V`. Which version of `V`'s constructor is executed and with what parameter? In fact, `V`'s default constructor is always executed. This would be the case even if `C1` and `C2` constructed their bases with the same integer parameter.

This is clarified by making the initialization explicit, as in:

```
D ( ) : C1 ( ), C2 ( ), V ( )
{
}
```

Rule 12–1–3	**(Required)**	**All constructors that are callable with a single argument of fundamental type shall be declared *explicit*.**

Rationale

The *explicit* keyword prevents the constructor from being used to implicitly convert from a fundamental type to the class type.

Example

```
class C
{
public:
   C ( int32_t a )            // Non-compliant
   {
   }
};
class D
{
public:
   explicit D ( int32_t a )   // Compliant
   {
   }
};
```

6.12.8 Copying class objects

Rule 12–8–1	**(Required)**	**A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.**

[Implementation 12.8(15)]

Rationale

If a compiler implementation detects that a call to a copy constructor is redundant, then it is permitted to omit that call, even if the copy constructor has a side effect other than to construct a copy of the object. This is called *copy elision*.

It is therefore important to ensure that a copy constructor does not modify the program state as the number of such modifications may be indeterminate.

Example

```
class A
{
public:
   A ( A const & rhs )
   : m_i ( rhs.m_i )
   {
     ++m_static;   // Non-compliant
   }
private:
          int32_t m_i;
   static int32_t m_static;
};
int32_t A::m_static = 0;
A f ( )
{
   return A ( );
}
```

```
void b ( )
{
   A a = f ( );
}
```

The value that `m_static` has after the call to `b( )` is *implementation-defined*.

Rule 12–8–2	**(Required)**	**The copy assignment operator shall be declared *protected* or *private* in an abstract class.**

Rationale

An abstract class represents the interface part of a hierarchy. Invoking the copy constructor from the top of such a hierarchy bypasses the underlying implementation resulting in only the base sub-objects being copied.

Example

```
class B1
{
public:
   B1 ( );
   virtual void f( ) = 0;
   B1 & operator= ( B1 const & rhs );   // Non-compliant
   int32_t getKind ( ) const { return kind; }
private:
   int32_t kind;
};
class D1 : public B1
{
public:
   virtual void f ( ) { }
   D1 & operator= ( D1 const & rhs );
private:
   int32_t member;
};
void f1( B1 & b1, B1 & b2 )
{
   b1 = b2;
}
```

As the assignment operator is public, the function `f1` can call the operator and so copies the base sub-objects of `b1` and `b2`. As the type of `b1` and `b2` is an abstract type, `b1` and `b2` must be sub-objects, and so the information contained in the derived objects for both will not be copied.

Making the abstract copy assignment operator protected allows access from the derived classes but not from outside the hierarchy.

```
class B2
{
public:
   B2 ( );
   virtual void f ( ) = 0;
   int32_t getKind ( ) const { return kind; }
```

```
protected:
   B2 & operator= ( B2 const & rhs );   // Compliant
private:
   int32_t kind;
};
class D2 : public B2
{
public:
   virtual void f ( ) { }
   D2 & operator= ( D2 const & rhs );
};
void f2 ( B2 & b1, B2 & b2 )
{
  b1 = b2;      // Compiler error will be reported
}
```

Making the copy assignment operator private is a common idiom used to restrict copying objects of the class type.

6.14 Templates

6.14.5 Template declarations

Rule 14–5–1	**(Required)**	**A non-member *generic function* shall only be declared in a namespace that is not an *associated namespace*.**

Rationale

Argument-dependent lookup (*ADL*) adds additional *associated namespaces* to the set of scopes searched when lookup is performed for the names of called functions. A *generic function* found in one of these additional namespaces would be added to the overload set and chosen by overload resolution, which is inconsistent with developer expectation.

Example

```
template <typename T>
class B
{
public:
   B operator+ ( long & rhs );
   void f ( )
   {
     *this + 10;   // calls NS::operator+ and not
                   // B<NS::A>::operator+ when B is
                   // instantiated with NS::A
   }
};
namespace NS
{
   class A {
   public:
   };
```

```
   template <typename T>
   bool operator+ ( T, int32_t );  // Non-compliant – within associated
                                   //                 namespace
}
template class B<NS::A>;
```

ADL considers the namespace `NS` to be an *associated namespace*. There are three functions in the overload set:

- The built-in `operator+`

  ```
  T operator+ ( T, T );
  ```

- The member `operator+`

  ```
  B<NS::A> B<NS::A>::operator+ ( long );
  ```

- The specialized *generic function*

  ```
  bool NS::operator+< B<NS::A> > ( B<NS::A>, int32_t )
  ```

The conversion from the literal 10 to `int32_t` is a better match than that to `long`, and therefore `NS::operator+` is chosen rather than the member `operator+`, which may be inconsistent with developer expectations.

Rule 14–5–2	**(Required)**	**A *copy constructor* shall be declared when there is a template constructor with a single parameter that is a *generic parameter*.**

Rationale

Contrary to possible developer expectations, a template constructor will not suppress the compiler generated *copy constructor*. This may lead to incorrect copy semantics for members requiring deep copies.

Example

```
class A
{
public:
   A ( );
   // A ( A const & rhs );   Example 1 - implicitly generated
   template <typename T>
   A ( T const & rhs )    // Example 2
   : i ( new int32_t )
   {
     *i = *rhs.i;
   }
private:
   int32_t * i;           // Member requires deep copy
};
void f ( A const & a1 )
{
   A a2 ( a1 );           // Non-compliant - Unexpectedly uses Example 1
}
```

The implicitly generated copy constructor, Example 1, will be used to construct `a2` from `a1`.

Therefore, a shallow copy on the pointer member `i` will result in both `a1.i` and `a2.i` pointing to the same object. Was this the intent, or was it expected that a new object would be created and initialized?

Rule 14–5–3	**(Required)**	**A *copy assignment operator* shall be declared when there is a template assignment operator with a parameter that is a *generic parameter*.**

Rationale

Contrary to possible developer expectations, a template assignment operator will not suppress the compiler generated *copy assignment operator*. This may lead to incorrect copy semantics for members requiring deep copies.

Example

```
class A
{
public:
   // A & operator= ( A const & rhs )   Example 1 - implicitly generated
   // {
   //    i = rhs.i;
   //    return *this;
   // }
   template <typename T>
   T & operator= ( T const & rhs )   // Example 2
   {
      if ( this != &rhs ) {
         delete i;
         i = new int32_t;
         *i = *rhs.i;
      }
      return *this;
   }
private:
   int32_t * i;  // Member requires deep copy
};
void f ( A const & a1, A & a2 )
{
   a2 = a1;   // Unexpectedly uses Example 1
}
```

The implicitly generated *copy assignment operator* Example 1 will be used to copy `a1` to `a2`. Therefore, a shallow copy on the pointer member `i` will result in both `a1.i` and `a2.i` pointing to the same object. Was this the intent, or was it expected that a new object would be created and initialized?

6.14.6 Name resolution

Rule 14–6–1	(Required)	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a *qualified-id* or `this->`

Rationale

Using a *qualified-id* or prefixing the identifier with `this->` ensures that the entity chosen is consistent with developer expectations.

Example

```
typedef int32_t TYPE;
void g ( );

template <typename T>
class B;

template <typename T>
class A : public B<T>
{
   void f1 ( )
   {
     TYPE t = 0;                  // Non-compliant Example 1
     g ( );                       // Non-compliant Example 2
   }

   void f2 ( )
   {
     ::TYPE t1 = 0;               // Compliant - explicit use global TYPE
     ::g ( );                     // Compliant - explicit use global func

     typename B<T>::TYPE t2 = 0; // Compliant - explicit use base TYPE
     this->g ( );                 // Compliant - explicit use base "g"
   }
};

template <typename T>
class B
{
public:
  typedef T TYPE;
  void g ( );
};

template class A<int32_t>;
```

A conforming compiler will choose `::TYPE` in Example 1, and `::g` in Example 2.

Rule 14–6–2	(Required)	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

Rationale

Argument-dependent lookup (*ADL*) adds additional *associated namespaces* to the set of scopes searched when lookup is performed for the names of called functions. For function templates, *ADL* is performed at the point of instantiation of the function template, and so it is possible that a

function declared after the template may be called.

To ensure that *ADL* does not take place when calling a function with a dependent argument, the *postfix-expression* denoting the called function can either be a qualified name or a parenthesized expression.

Example

```
void b ( int32_t );

template <typename T>
void f ( T const & t )
{
   b ( t );           // Non-compliant - Calls NS::b declared after f
   ::b ( t );         // Compliant - Calls ::b
   ( b ) ( t );       // Compliant - Calls ::b
}

namespace NS
{
   struct A
   {
      operator int32_t ( ) const;
   };
   void b ( A const & a );
}

int main ( )
{
   NS::A a;
   f ( a );
}
```

Operators with dependent types may also have this problem. In order to avoid *ADL* in these examples, operators should not be overloaded, or the calls should be changed to use explicit function call syntax and a qualified name or parenthesized expression used, as above.

For example:

```
template <typename T>
void f ( T const & t )
{
   t == t;                       // Non-compliant - Calls NS::operator==
                                 // declared after f
   ::operator ==( t, t );        // Compliant - Calls built-in operator==
   ( operator == <T> ) ( t, t ); // Compliant - Calls built-in operator==
}

namespace NS
{
   struct A
   {
      operator int32_t ( ) const;
   };
   bool operator== ( A const &, A const & );
}
```

```
int main ( )
{
   NS::A a;
   f ( a );
}
```

6.14.7 Template instantiation and specialization

Rule 14–7–1	(Required)	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.

Rationale

Similar to uncalled functions, un-instantiated class and function templates are a potential source of noise and they may be symptomatic of a more serious problem such as missing paths.

Note: Even though a given class template may be instantiated many times, it is possible that some of its member functions are never instantiated.

See Section 3.5 for associated library issues.

Example

```
template < typename T >
class Sample
{
public:
   void inst_mem ( )
   {
      ...
   }
   void uninst_mem ( )   // Non-compliant
   {
      ...
   }
};
Sample<int64_t> s;
s.inst_mem ( );     // Call to s.inst_mem instantiates the member.
                    // s.uninst_mem is not called within the program
                    // and is not instantiated.
```

Rule 14–7–2	(Required)	For any given template specialization, an explicit instantiation of the template with the *template-arguments* used in the specialization shall not render the program ill-formed.

Rationale

An implicit template specialization does not instantiate every member of the template. Where instantiation of a member would result in an *ill-formed* program it is not clear that the template should be used with the supplied *template-arguments*.

See Section 3.5 for associated library issues.

Example

```
template <typename T>
class A
{
public:
   void f1 ()
   {
      // ...
   }
   void f2 ()
   {
      T t;
      t.x = 0;         // Will only work for types that have a .x member
   }
};
void b ()
{
   A<int32_t> a;       // A<int32_t>::f2 is not instantiated.
   a.f1 ();
}
template class A<int32_t>; // Non-compliant - instantiation of f2
                           // results in "ill-formed" program.
```

Rule 14–7–3	(Required)	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their *primary template*.

[NDR 14.5.4(1), 14.6.4.1(7), 14.7.3(6)]

Rationale

It is *undefined behaviour* if, for a set of *template-arguments*, an implicit instantiation is generated by the compiler, and a partial or explicit specialization is declared or defined elsewhere in the program that would match the set of *template-arguments*.

Example

```
// tmpl.h
template <typename T> void bad_tmpl () {}
template <typename T> void good_tmpl () {}
template <> void good_tmpl<int32_t> () {}
// tmpl.cc
#include "tmpl.h"
template <> void bad_tmpl<int32_t> () {}    // Non-compliant
template <> void good_tmpl<int32_t> () {}
// f.cc
#include <tmpl.h>
```

```
void f ()
{
   bad_tmpl<int32_t> ();  // implicit instantiation of primary.
                          // explicit instantiation in tmpl.cc would
                          // have been used if it were visible.
   good_tmpl<int32_t> (); // specialization of good_tmpl<int32_t> is
                          // visible with the primary declaration.
}
```

6.14.8 Function template specialization

Rule 14–8–1	(Required)	Overloaded function templates shall not be explicitly specialized.

Rationale

Explicit specializations will be considered only after overload resolution has chosen a best match from the set of primary function templates. This may be inconsistent with developer expectations.

Example

```
template <typename T> void f ( T );  // overload Example 1
template <typename T> void f ( T* ); // overload Example 2

template <> void f<int32_t*> ( int32_t* ); // explicit specialization of
                                           // Example 1

void b ( int32_t * i )
{
   f ( i );                      // Non-compliant
                                 // - Calls Example 2, f<int32_t*>
}
```

Where a template is not overloaded with other templates, or is overloaded with non-template functions then it can be explicitly specialized, as it is consistent with developer expectation that the explicit specializations will only be considered if that *primary template* is chosen.

```
template <typename T> void f ( T );          // Example 1
template <> void f<int32_t*> ( int32_t* );   // Example 2

void b ( int32_t * i )
{
   f ( i );                          // Compliant
                                     // - Calls Example 2, f<int32_t*>
}
```

Rule 14–8–2	(Advisory)	The viable *function set* for a function call should either contain no function specializations, or only contain function specializations.

Rationale

If a function and a specialization of a function template are deemed equivalent after overload resolution, the non-specialized function will be chosen over the function specialization, which may be inconsistent with developer expectations.

Exception

This rule does not apply to copy constructors or copy assignment operators.

Example

```
void f ( short );                        // Example 1
template <typename T> void f ( T );      // Example 2

void b ( short s )
{
   f ( s );         // Non-compliant - Calls Example 1
   f ( s + 1 );     // Non-compliant - Calls Example 2

   f<>( s );        // Compliant - Explicitly calls Example 2
   f<>( s + 1 );    // Compliant - Explicitly calls Example 2
}
```

6.15 Exception handling

6.15.0 General

Rule 15–0–1 (Document) Exceptions shall only be used for error handling.

Rationale

The exception handling mechanism can provide an effective and clear means of handling error conditions, particularly where a function needs to return both some desired result together with an indication of success or failure. However, because of its ability to transfer control back up the call tree, it can also lead to code that is difficult to understand. Hence it is required that the mechanism is only used to capture behaviour that is in some sense undesirable, and which is not expected to be seen in normal program execution.

Example

The `try...catch` mechanism is an acceptable means of catching error conditions thrown locally or in any called functions.

For example, a file reader may fail in a number of different ways: failure to open the file, unexpected or disallowed data in the file, insufficient data in the file, etc. The requirement to process data up to the point where an error is detected, then to perform some tidy up action (or return some success indicator if no error is found), could be achieved, either using `goto` (undesirable for the reasons described in Rule 6–6–1) or status flags and repeated nested tests to skip further processing once an error is detected. It is argued that the `try...catch` mechanism provides a clearer way of separating the expected and exceptional behaviours.

```
bool ReadFile ( const char *name )
{
   try
   {
      if ( /* open name fails */ )
      {
         throw ( "failed to open file" );
      }
```

```
            if ( /* unexpected data */ )
            {
               throw ( "unexpected data found in file" );
            }
            return true;
         }
         catch ( const char *message )
         {
            // tidy up after failure to read
            return false;
         }
      }
```

The following:

```
      void fn ( )
      {
         try
         {
            if ( x < 10 )
            {
               throw ( 10 );
            }
            // Action "A"
         }
         catch ( int32_t y )
         {
            // Action "B"
         }
      }
```

could be used as an expensive and confusing way of implementing

```
      if ( x < 10 )
      {
         // Action "B"
      }
      else
      {
         // Action "A"
      }
```

Rule 15–0–2 (Advisory) An exception object should not have pointer type.

Rationale

If an exception object of pointer type is thrown and that pointer refers to a dynamically created object, then it may be unclear which function is responsible for destroying it, and when. This ambiguity does not exist if the object is caught by value or reference.

Example

```
class A
{
   // Implementation
};
void fn ( int16_t i )
{
   static A   a1;
          A * a2 = new A;
   if ( i > 10 )
   {
      throw ( &a1 );  // Non-compliant – pointer type thrown
   }
   else
   {
      throw ( a2 );   // Non-compliant – pointer type thrown
   }
}
```

Rule 15–0–3	(Required)	**Control shall not be transferred into a *try* or *catch* block using a *goto* or a *switch* statement.**

Rationale

A program is ill-formed if control is transferred into a *try* or *catch* block using a *goto* or *switch* statement; however, not all compilers issue a diagnostic message.

Example

```
void f ( int32_t i )
{
   if ( 10 == i )
   {
      goto Label_10;   // Non-compliant
   }
   if ( 11 == i )
   {
      goto Label_11;   // Non-compliant
   }
   switch ( i )
   {
      case 1:
         try
         {
            Label_10:
            case 2:        // Non-compliant – also violates switch rules
               // Action
               break;
         }
```

```
            catch ( ... )
            {
               Label_11:
               case 3:        // Non-compliant – also violates switch rules
                  // Action
                  break;
            }
            break;
         default:
         {
            // Default Action
            break;
         }
      }
   }
```

6.15.1 Throwing an exception

Rule 15–1–1	**(Required)**	**The *assignment-expression* of a *throw* statement shall not itself cause an exception to be thrown.**

Rationale

If an exception is thrown when constructing the exception object, or when evaluating the assignment expression that initializes the exception object, it is that exception that propagates in preference to the one that was about to be thrown. This may be inconsistent with developer expectations.

Example

```
class E
{
   public:
      E ( ){ }           // Assume constructor cannot cause an exception
};
try
{
   if ( ... )
   {
      throw E ( );       // Compliant – no exception thrown
   }                     //   when constructing the object
}
// construction of E2 causes an exception to be thrown
class E2
{
   public:
      E2 ( )
      {
         throw 10;
      }
};
```

```
try
{
   if ( ... )
   {
      throw E2 ( );  // Non-compliant – int exception thrown
                     // when constructing the E2 object
   }
}
```

Rule 15–1–2 (Required) *NULL* shall not be thrown explicitly.

Rationale

`throw(NULL)` (equivalent to `throw(0)`) is never a throw of the *null-pointer-constant* and so is only ever caught by an integer handler. This may be inconsistent with developer expectations, particularly if the program only has handlers for pointer-to-type exceptions.

Example

```
try
{
   throw ( NULL );       // Non-compliant
}
catch ( int32_t i )      // NULL exception handled here
{
   // ...
}
catch ( const char_t * ) // Developer may expect it to be caught here
{
   // ...
}
char_t * p = NULL;
try
{
   throw ( static_cast < const char_t * > ( NULL ) ); // Compliant,
                                                      // but breaks
                                                      // Rule 15-0-2
   throw ( p );                                       // Compliant
}
catch ( int32_t i )
{
   // ...
}
catch ( const char_t * )          // Both exceptions handled here
{
   // ...
}
```

Rule 15–1–3	(Required)	An empty *throw* (`throw;`) shall only be used in the *compound-statement* of a *catch* handler.

[Implementation 15.3(9), 15.5.1(2)]

Rationale

An empty *throw* re-throws the temporary object that represents an exception. Its use is intended to enable the handling of an exception to be split across two or more handlers.

However, syntactically, there is nothing to prevent `throw;` being used outside a catch handler, where there is no exception object to re-throw. This may lead to an *implementation-defined* program termination.

Example

```
void f1 ( void )
{
   try
   {
      throw ( 42 );
   }
   catch ( int32_t i )   // int will be handled here first
   {
      if ( i > 0 )
      {
         throw;          // and then re-thrown - Compliant
      }
   }
}
void g1 ( void )
{
   try
   {
      f1 ( );
   }
   catch ( int32_t i )
   {
      // Handle re-throw from f1 ( )
      // after f1's handler has done what it needs
   }
}
void f2 ( void )
{
   throw;   // Non-compliant
}
void g2 ( void )
{
   try
   {
      throw;   // Non-compliant
   }
```

```
   catch ( ... )
   {
      // ...
   }
}
```

6.15.3 Handling an exception

Rule 15–3–1	(Required)	Exceptions shall be raised only after start-up and before termination of the program.

[Implementation 15.3(9), 15.5.1(2)]

Rationale

Throwing an exception during start-up or termination results in the program being terminated in an *implementation-defined* manner.

Before the program starts executing the body of *main*, it is in a start-up phase, constructing and initializing static objects. Similarly, after *main* has returned, it is in a termination phase where static objects are being destroyed. If an exception is thrown in either of these phases it leads to the program being terminated in an *implementation-defined* manner. Specifically from Section 15.3(13) of ISO/IEC 14882:2003 [1], if *main* is implemented as a *function-try-block*, exceptions raised during start-up and close down are not caught by the *catch* block(s) of *main*.

This is in effect a special case of Rule 15–3–4, as there is nowhere a handler can be placed to catch exceptions thrown during start-up or termination.

Example

```
class C
{
public:
   C ( )
   {
      throw ( 0 );   // Non-compliant – thrown before main starts
   }
  ~C ( )
   {
      throw ( 0 );   // Non-compliant – thrown after main exits
   }
};
C c;   // An exception thrown in C's constructor or destructor will
       // cause the program to terminate, and will not be caught by
       // the handler in main
int main( ... )
{
   try
   {
      // program code
      return 0;
   }
```

```
   // The following catch-all exception handler can only
   // catch exceptions thrown in the above program code
   catch ( ... )
   {
      // Handle exception
      return 0;
   }
}
```

See also

Rule 15–3–3, Rule 15–3–4, Rule 15–5–1, Rule 15–5–3

Rule 15–3–2	(Advisory)	There should be at least one exception handler to catch all otherwise unhandled exceptions

[Implementation 15.3(9), 15.5.1(2)]

Rationale

If a program throws an unhandled exception it terminates in an *implementation-defined* manner. In particular, it is *implementation-defined* whether the call stack is unwound, before termination, so the destructors of any automatic objects may or may not be executed. By enforcing the provision of a "last-ditch catch-all", the developer can ensure that the program terminates in a consistent manner.

The objective of Rule 15–3–4 is that a program should catch all exceptions that it is expected to throw. This rule's objective is to ensure that exceptions that were not expected are also caught.

Example

For the majority of programs this will mean *main* should look like:

```
int32_t main( )
{
   try
   {
      // program code
   }
   catch ( ... )  // Catch-all handler
   {
      // Handle unexpected exceptions
   }
   return 0;
}
```

See also

Rule 15–3–4, Rule 15–5–3

Rule 15–3–3	(Required)	Handlers of a *function-try-block* implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

[Undefined 15.3(10)]

Rationale

The effect of accessing a non-static member of a class or a base class in the handler (i.e. the *catch* part) of a *function-try-block* of a class constructor/destructor is undefined.

For example, if a memory allocation exception is thrown during creation of the object, the object will not exist when the handler attempts to access its members. Conversely, in the destructor, the object may have been successfully destroyed before the exception is handled, so again will not be available to the handler.

By contrast, the lifetime of a static member is greater than that of the object itself, so the static member is guaranteed to exist when the handler accesses it.

Example

```
class C
{
public:
   int32_t x;
   C ( )
      try
      {
         // Action that may raise an exception
      }
      catch ( ... )
      {
         if ( 0 == x )  // Non-compliant – x may not exist at this point
         {
            // Action dependent on value of x
         }
      }
  ~C ( )
      try
      {
         // Action that may raise an exception
      }
      catch ( ... )
      {
         if ( 0 == x )  // Non-compliant – x may not exist at this point
         {
            // Action dependent on value of x
         }
      }
};
```

See also

Rule 15–3–1, Rule 15–5–1

Rule 15–3–4	(Required)	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

[Implementation 15.3(9), 15.5.1(2)]

Rationale

If a program throws an unhandled exception, it terminates in an *implementation-defined* manner. In particular, it is *implementation-defined* whether the call stack is unwound before termination, so the destructors of any automatic objects may or may not be invoked.

If an exception is thrown as an object of a derived class, a "compatible type" may be either the derived class or any of its bases.

The objective of this rule is that a program should catch all exceptions that it is expected to throw. The objective of Rule 15–3–2 is to ensure that exceptions that were not expected are also caught.

Example

```
class A {};
class B {};

void f ( int32_t i ) throw ( )
{
   try
   {
      if ( i > 10 )
      {
         throw A ( );  // Compliant
      }
      else
      {
         throw B ( );  // Non-compliant
      }
   }
   catch ( A const & )
   {
   }
}
```

See also

Rule 15–3–2, Rule 15–5–3

Rule 15–3–5	(Required)	A class type exception shall always be caught by reference.

Rationale

If a class type exception object is caught by value, slicing occurs. That is, if the exception object is of a derived class and is caught as the base, only the base class's functions (including virtual functions) can be called. Also, any additional member data in the derived class cannot be accessed.

If the exception is caught by reference, slicing does not occur.

Example

```
// base class for exceptions
class ExpBase
{
public:
   virtual const char_t *who ( )
   {
      return "base";
   };
};
class ExpD1: public ExpBase
{
public:
   virtual const char_t *who ( )
   {
      return "type 1 exception";
   };
};
class ExpD2: public ExpBase
{
public:
   virtual const char_t *who ( )
   {
      return "type 2 exception";
   };
};
try
{
   // ...
   throw ExpD1 ( );
   // ...
   throw ExpBase ( );
}
catch ( ExpBase &b )   // Compliant – exceptions caught by reference
{
   // ...
   b.who();  // "base", "type 1 exception" or "type 2 exception"
             // depending upon the type of the thrown object
}
// Using the definitions above ...
catch ( ExpBase b ) // Non-compliant - derived type objects will be
                    // caught as the base type
{
   b.who();         // Will always be "base"
   throw b;         // The exception re-thrown is of the base class,
                    // not the original exception type
}
```

Rule 15–3–6	(Required)	**Where multiple handlers are provided in a single *try-catch* statement or *function-try-block* for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.**

Rationale

When testing to see if the type of an exception matches the type of a handler, a derived class exception will match with a handler for its base class. If the base class handler is found before the handler for the derived class, the base class handler will be used. The derived class handler is *unreachable code* and can never be executed.

Example

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

try
{
   // ...
}
catch ( D &d )   // Compliant – Derived class caught before base class
{
   // ...
}
catch ( B &b )   // Compliant – Base class caught after derived class
{
   // ...
}
// Using the classes from above ...
try
{
   // ...
}
catch ( B &b )   // Non-compliant – will catch derived classes as well
{
   // ...
}
catch ( D &d )   // Non-compliant – Derived class will be caught above
{
                 // Any code here will be unreachable,
                 // breaking Rule 0–1–1
}
```

Rule 15–3–7	(Required)	**Where multiple handlers are provided in a single *try-catch* statement or *function-try-block*, any ellipsis (catch-all) handler shall occur last.**

Rationale

If the catch-all handler is found before any other handler, that behaviour will be performed. Any handlers after the catch-all are *unreachable code* and can never be executed.

Example

```
void f1 ( )
{
   try
   {
      // ...
   }
   catch ( int32_t i )   // int handler
   {
      // Handle int exceptions
   }
   catch( ... )          // catch-all handler
   {
      // Handle all other exception types
   }
}
void f2 ( )
{
   try
   {
      // ...
   }
   catch( ... )          // catch-all handler
   {
      // Handle all exception types
   }
   catch ( int32_t i )   // Non-compliant – handler will never be called
   {
   }
}
```

6.15.4 Exception specifications

Rule 15–4–1	**(Required)**	**If a function is declared with an *exception-specification*, then all declarations of the same function (in other translation units) shall be declared with the same set of *type-ids*.**

[NDR 15.4(2)]

Rationale

It is *undefined behaviour* if a function has different *exception-specifications* in different translation units.

Example

```
// Translation unit A
void foo( ) throw ( const char_t * )
{
   throw "Hello World!";
}
// Translation unit B
// foo declared in this translation unit with a different exception
// specification
extern void foo ( ) throw ( int32_t );   // Non-compliant -
                                         // different specifier

void b ( ) throw ( int32_t )
{
   foo ( );   // The behaviour here is undefined.
}
```

6.15.5 Special functions

Rule 15–5–1	**(Required)**	**A class destructor shall not exit with an exception.**

[Implementation 15.3(9), 15.5.1(2)]

Rationale

When an exception is thrown, the call stack is unwound up to the point where the exception is to be handled. The destructors for all automatic objects declared between the point where the exception is thrown and where it is to be handled will be invoked. If one of these destructors exits with an exception, then the program will terminate in an *implementation-defined* manner.

Note that it is acceptable for a destructor to throw an exception that is handled within the destructor, for example within a *try-catch* block.

Example

```
class C1
{
public:
   ~C1 ( )
   {
      try
      {
         throw ( 42 );      // Compliant - exception will not leave
                            // destructor
      }
      catch ( int32_t i )   // int handler
      {
         // Handle int exception throw by destructor
      }
   }
};
class C2
{
public:
   ~C2 ( )
   {
      throw ( 42 );      // Non-compliant - destructor exits with an
                         // exception
   }
};
void foo ( )
{
   C2 c; // program terminates when c is destroyed
   throw ( 10 );
}
```

See also

Rule 15–3–1, Rule 15–3–3, Rule 15–5–3

Rule 15–5–2	**(Required)**	**Where a function's declaration includes an *exception-specification*, the function shall only be capable of throwing exceptions of the indicated type(s).**

[Implementation 15.5.1(2)]

Rationale

If a function declared with an *exception-specification* throws an exception of a type not included in the specification, the function *unexpected()* is called. The behaviour of this function can be overridden within a *project*, but by default causes an exception of *std::bad_exception* to be thrown. If *std::bad_exception* is not listed in the *exception-specification*, then *terminate()* will be called, leading to *implementation-defined* termination of the program.

Example

```
// foo does not have an exception specification, so can propagate
// exceptions of any type, including int
void foo ( )
{
   throw ( 21 );
}
// goo specifies that it will only throw exceptions of type Exception.
// If foo throws an int the function unexpected() is called, which may
// terminate the program
void goo ( ) throw ( Exception )
{
   foo ( ); // Non-compliant – int is not listed in the
            // throw specification
}
```

Rule 15–5–3	(Required)	The *terminate()* function shall not be called implicitly.

[Implementation 15.5.1(2)]

Rationale

It is *implementation-defined* whether the call stack is unwound before *terminate()* is called, so the destructors of any automatic objects may or may not be executed.

See also

Rule 15–3–1, Rule 15–3–2, Rule 15–3–4, Rule 15–5–1. The situations addressed by these rules cause the program to call *terminate()* and so exhibit *implementation-defined behaviour*.

6.16 Preprocessing directives

6.16.0 General

Rule 16–0–1	(Required)	*#include* directives in a file shall only be preceded by other preprocessor directives or comments.

Rationale

To aid code readability, all the *#include* directives in a particular code file should be grouped together near the head of the file. The only items which may precede a *#include* in a file are other preprocessor directives or comments.

Example

```
#include <f1.h>    // Compliant
int32_t i;
#include <f2.h>    // Non-compliant
```

Rule 16–0–2	(Required)	Macros shall only be *#define*'d or *#undef*'d in the global namespace.

Rationale

While it is legal to place *#define* or *#undef* directives anywhere in a source file, placing them outside of the global namespace is misleading as their scope is not restricted. This may be inconsistent with developer expectations.

Example

```
#ifndef MY_HDR
#define MY_HDR       // Compliant

namespace NS
{
   #define FOO       // Non-compliant
   #undef FOO        // Non-compliant
}

#endif
```

See also

Rule 16–0–3

Rule 16–0–3	(Required)	*#undef* shall not be used.

Rationale

#undef should not normally be needed. Its use can lead to confusion with respect to the existence or meaning of a macro when it is used in the code.

Example

```
#ifndef MY_HDR
#define MY_HDR

#undef MY_HDR        // Non-compliant

#endif
```

Rule 16–0–4	(Required)	Function-like macros shall not be defined.

[Undefined 16.3(10)]

Rationale

While macros can provide a speed advantage over functions, functions provide a safer and more robust mechanism. This is particularly true with respect to the type checking of parameters, and the problem of function-like macros potentially evaluating parameters multiple times.

Inline functions should be used instead.

Example

```
#define FUNC_MACRO(X) ((X)+(X))       // Non-compliant
```

Rule 16–0–5	(Required)	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

[Undefined 16.3(10)]

Rationale

If any of the arguments act like preprocessor directives, the behaviour when macro substitution is made can be unpredictable.

Example

```
#define M(A) printf ( #A )
void main ( )
{
   M(
#ifdef SW        // Non-compliant
   "Message 1"
#else            // Non-compliant
   "Message 2"
#endif           // Non-compliant
    );
}
```

The above may print

```
#ifdef SW "Message 1" #else "Message 2" #endif
```

or

```
Message 2
```

Rule 16–0–6	(Required)	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Rationale

If parentheses are not used, then the operator precedence may not give the desired results when the preprocessor substitutes the macro into the code.

Within a definition of a function-like macro, the arguments shall be enclosed in parentheses.

Example

Define an `abs` function using:

```
#define abs(x) (((x) >= 0) ? (x) : -(x))   // Compliant
```

and not:

```
#define abs(x) ((x >= 0) ? x : -x)         // Non-compliant
```

Consider what happens if the second, incorrect, definition is substituted into the expression:

```
z = abs( a - b );
```

giving:

```
z = ((a - b >= 0) ? a - b : -a – b);
```

The sub-expression `-a - b` is equivalent to `(-a)-b` rather than `-(a-b)` as intended. Putting all the parameters in parentheses in the macro definition avoids this problem.

```
#define subs(x) a ## x   // Compliant
```

Rule 16–0–7	(Required)	**Undefined macro identifiers shall not be used in *#if* or *#elif* preprocessor directives, except as operands to the *defined* operator.**

Rationale

If an attempt is made to use an identifier in a preprocessor directive, and that identifier has not been defined, the preprocessor will assume the value zero. *#ifdef*, *#ifndef* and *defined()* are provided to test the existence of a macro, and are therefore excluded.

Example

```
#if x < 0   // Non-compliant - x assumed to be zero as it is not defined
```

Consideration should be given to the use of a *#ifdef* test before an identifier is used.

Note that preprocessing identifiers may be defined either by use of *#define* directives or by options specified at compiler invocation. However, the use of the *#define* directive is preferred.

Rule 16–0–8	(Required)	**If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.**

Rationale

When a section of source code is excluded by preprocessor directives, the content of each excluded statement is ignored until a *#else*, *#elif* or *#endif* directive is encountered (depending on the context). If one of these excluded directives is badly formed, it may be ignored without warning by a compiler with unexpected consequences.

The requirement of this rule is that all preprocessor directives shall be syntactically valid even when they occur within an excluded block of code.

In particular, ensure that *#else* and *#endif* directives are not followed by any characters other than white-space. Compilers are not always consistent in enforcing this requirement.

Example

```
#define AAA 2
int32_t foo(void)
{
   int32_t x = 0;
   ...
#ifndef AAA
   x = 1;
#else1                  // Non-compliant
   x = AAA;
#endif
   ...
   return x;
}
```

6.16.1 Conditional inclusion

Rule 16–1–1	(Required)	The *defined* preprocessor operator shall only be used in one of the two standard forms.

[Undefined 16.1(4)]

Rationale

The only two permissible forms for the *defined* preprocessor operator are:

```
defined ( identifier )
defined identifier
```

Any other form is a constraint violation, but this is not reported by all compliers.

Example

```
#if defined ( X > Y )   // Non-compliant – constraint violation
```

Generation of the token defined during expansion of a *#if* or *#elif* preprocessing directive also leads to *undefined behaviour* and shall be avoided, for example:

```
#define DEFINED defined
#if DEFINED(X)          // Non-compliant - undefined behaviour
```

Rule 16–1–2	(Required)	All *#else*, *#elif* and *#endif* preprocessor directives shall reside in the same file as the *#if* or *#ifdef* directive to which they are related.

Rationale

When the inclusion and exclusion of blocks of statements is controlled by a series of preprocessor directives, confusion can arise if all of the relevant directives do not occur within one file. This rule requires that all preprocessor directives in a sequence of the form *#if* / *#ifdef* ... *#elif* ... *#else* ... *#endif* shall reside in the same file. Observance of this rule preserves good code structure and avoids maintenance problems.

Notice that this does not preclude the possibility that such directives may exist within included files provided that all directives that relate to the same sequence are located in one file.

Example

```
// file.cpp
#define A
...
#ifdef A
...
#include "file1.hpp"
#
#endif
...
#if 1
#include "file2.hpp"
...
```

```
// file1.hpp
#if 1
...
#endif   // Compliant
// file2.hpp
...
#endif   //  Non-compliant
```

6.16.2 Source file inclusion

Rule 16–2–1	(Required)	The pre-processor shall only be used for file inclusion and *include guards*.

Rationale

C++ provides safer ways of achieving what is often done using the pre-processor, by way of inline functions and constant declarations.

Example

```
#ifndef HDR          // Compliant
#define HDR          // Compliant
#define X(Y) (Y)     // Non-compliant
#endif
```

See also

Rule 16–2–2

Rule 16–2–2	(Required)	C++ macros shall only be used for *include guards*, type qualifiers, or storage class specifiers.

Rationale

These are the only permitted uses of macros. C++ offers const variable and function templates, which provide a type-safe alternative to the preprocessor.

Note that the use of macros for type qualifiers and storage class specifiers will break Rule 16–2–1.

Example

```
// The following are compliant
#define STOR extern                     // storage class specifier
                                        // Breaks Rule 16-2-1

// The following are non-compliant
#define CLOCK (xtal/16)                 // Constant expression
#define PLUS2(X) ((X) + 2)              // Macro expanding to expression
#define PI 3.14159F                     // use const object instead
#define int32_t long                    // use typedef instead
#define STARTIF if(                     // language redefinition
#define INIT(value) {(value), 0, 0}     // braced initializer
#define HEADER "filename.h"             // string literal
```

See also

Rule 16–2–1

Rule 16–2–3 (Required) *Include guards* shall be provided.

The *include guard* shall use one of the following two forms:

```
<start-of-file>
// Comments allowed here
#if !defined ( identifier )
#define identifier
   // Contents of file
#endif
<end-of-file>
```

```
<start-of-file>
// Comments allowed here
#ifndef identifier
#define identifier
   // Contents of file
#endif
<end-of-file>
```

Rationale

When a translation unit contains a complex hierarchy of nested *header files*, it is possible for a particular *header file* to be included more than once. This can be, at best, a source of confusion. If this multiple inclusion leads to multiple or conflicting definitions, then this can result in undefined or erroneous behaviour.

These forms are mandated to facilitate checking.

Example

```
// file.h
#ifndef FILE_H
#endif
```

```
// file.cc
#include "file.h"
#define FILE_H          // Non-compliant
```

Rule 16–2–4 (Required) The `'`, `"`, `/*` or `//` characters shall not occur in a *header file* name.

[Undefined 2.8(2)]

Rationale

It is *undefined behaviour* if the `'`, `"`, `/*` or `//` characters are used between < and > delimiters or the `'`, `/*` or `//` characters are used between the `"` delimiters in a header name preprocessing token.

Example

```
#include "fi'le.h"    // Non-compliant
```

Rule 16–2–5	(Advisory)	The \ character should not occur in a *header file* name.

[Undefined 2.8(2)]

Rationale

It is *undefined behaviour* if the \ character is used between < and > delimiters or between the " delimiters in a header name preprocessing token.

Note that this rule is only advisory, since some environments use \ as a file name delimiter. Compilers for these environments often support the use of / in *#include* directives.

Example

```
#include "fi\\le.h"    // Non-compliant
```

Rule 16–2–6	(Required)	The *#include* directive shall be followed by either a *<filename>* or *"filename"* sequence.

[Undefined 16.2(4)]

Rationale

These are the only forms for the *#include* directive permitted by ISO/IEC 14882:2003 [1].

Example

```
#include "filename.h"          // Compliant
#include <filename.h>          // Compliant
#define HEADER "filename.h"    // Non-compliant with Rule 16-2-2
#include HEADER                // Compliant
#include another.h            // Non-compliant
```

6.16.3 Macro replacement

Rule 16–3–1	(Required)	There shall be at most one occurrence of the # or ## operators in a single macro definition.

[Unspecified 16.3.2(2), 16.3.3(3), Undefined 16.3.2(2), 16.3.3(3)]

Rationale

The order of evaluation associated with both the # and ## preprocessor operators is unspecified. This problem can be avoided by having only one occurrence of either operator in any single macro definition (i.e. one #, **or** one ## **or** neither).

Example

```
#define A(x)          #x           // Compliant
#define B(x, y)       x ## y       // Compliant
#define C(x, y)       # x ## y     // Non-compliant
```

In the following, if y is joined to z first then the fourth parameter of D will be substituted and joined to x. Alternatively, if x is joined with y first, then the fourth parameter of D will not be substituted.

```
#define D(x, y, z, yz) x ## y ## z   // Non-compliant
```

Rule 16–3–2	(Advisory)	The # and ## operators should not be used.

[Unspecified 16.3.2(2), 16.3.3(3), Undefined 16.3.2(2), 16.3.3(3)]

Rationale

The order of evaluation associated with both the # and ## preprocessor operators is unspecified. Compilers have been known to implement these operators inconsistently, therefore, to avoid these problems, do not use them.

Example

```
#define A(Y)   #Y      // Non-compliant
#define A(X,Y) X##Y    // Non-compliant
```

6.16.6 Pragma directive

Rule 16–6–1	(Document)	All uses of the *#pragma* directive shall be documented.

[Implementation 16.6(1)]

Rationale

The *#pragma* directive is *implementation-defined*, hence it is important to demonstrate that all uses are correct.

This rule places a requirement on the user of this document to produce a list of any *pragmas* they choose to use in an application. The meaning of each *pragma* shall be documented. There shall be sufficient supporting description to demonstrate that the behaviour of the *pragma* and its implications for the application, have been fully understood.

Any use of *pragmas* should be minimized, localized and encapsulated within dedicated functions wherever possible.

6.17 Library introduction

6.17.0 General

Rule 17–0–1	(Required)	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

[Undefined 16.8(3)]

Rationale

It is generally bad practice to *#undef* a macro that is defined in the standard library. It is also bad practice to *#define* a macro name that is a C++ reserved identifier, or C++ keyword or the name of any macro, object or function in the standard library. For example, there are some specific reserved words and function names that are known to give rise to *undefined behaviour* if they are redefined or undefined, including *defined*, *_ _LINE_ _*, *_ _FILE_ _*, *_ _DATE_ _*, *_ _TIME_ _*, *_ _STDC_ _*, *errno* and *assert*.

Refer to ISO/IEC 14882:2003 [1] for a list of the identifiers that are reserved. Generally, all identifiers that begin with the underscore character are reserved.

Note that this rule applies regardless of which *header files*, if any, are actually included.

Example

```
#define __TIME__ 11111111    // Non-compliant
```

See also

Rule 16–0–3

Rule 17–0–2	(Required)	The names of standard library macros and objects shall not be reused.

Rationale

Where the developer uses new versions of standard library macros or objects (e.g. to enhance functionality or add checks of input values), the modified macro or object shall have a new name. This is to avoid any confusion as to whether a standard macro or object, or a modified version of them, is being used.

Example

```
#define NULL ( a > b )   // Non-compliant
```

Rule 17–0–3	(Required)	The names of standard library functions shall not be overridden.

Rationale

Where the developer uses new versions of standard library functions (e.g. to enhance functionality or add checks of input values), the modified function shall have a new name. However, it is permissible to overload the name to add new parameter types if the functionality is consistent with those of the original. This ensures that the behaviour associated with the name remains consistent. So, for example, if a new version of the *sqrt* function is written to check that the input is not negative, the new function shall not be named “sqrt”, but shall be given a new name. It is permissible to add a new *sqrt* function for a type not present in the library.

Example

```
int32_t printf ( int32_t a, int32_t b )   // Non-compliant
{
   return ( ( a > b ) ? a : b );
}
```

Rule 17–0–4	(Document)	All library code shall conform to MISRA C++.

Rationale

The quality of any libraries (linked or included) must be to at least the same standard (i.e. SIL level) as the rest of the *project*.

Library code may exist either as source code or object code. Either type shall include documentation to demonstrate how that code complies with MISRA C++.

Rule 17–0–5	(Required)	The *setjmp* macro and the *longjmp* function shall not be used.

Rationale

setjmp and *longjmp* allow the normal function call mechanisms to be bypassed, and shall not be used, since exception handling provides a better defined mechanism for this.

Example

```
#include <setjmp.h>
void f2 ( );
jmp_buf buf;
void f1 ( )
{
   if ( !setjmp ( buf ) )   // Non-compliant
   {
      f2 ( );
   }
   else
   {
   }
}
void f2 ( )
{
   longjmp ( buf, 10 );    // Non-compliant
}
```

6.18 Language support library

6.18.0 General

Rule 18–0–1	(Required)	The C library shall not be used.

Rationale

Some C++ libraries (e.g. <*cstdio*>) also have corresponding C versions (e.g. <*stdio.h*>). This rule requires that the C++ version is used.

Rule 18–0–2	(Required)	The library functions *atof*, *atoi* and *atol* from library <*cstdlib*> shall not be used.

Rationale

These functions have *undefined behaviour* associated with them when the string cannot be converted.

Example

```
#include <cstdlib>
```

```
int32_t f ( const char_t * numstr )
{
   return atoi ( numstr );    // Non-compliant
}
```

See also

ISO/IEC 9899:1990 [16] §7.10.1

Rule 18–0–3	(Required)	The library functions *abort, exit, getenv* and *system* from library *<cstdlib>* shall not be used.

Rationale

The use of these functions leads to *implementation-defined behaviour*.

Example

```
#include <cstdlib>
void f ( )
{
   exit ( 0 );      // Non-compliant
}
```

Rule 18–0–4	(Required)	The time handling functions of library *<ctime>* shall not be used.

Rationale

Various aspects are *implementation-defined* or *unspecified*, such as the formats of times.

Example

```
#include <ctime>
void f ( )
{
   clock ( );         // Non-compliant
}
```

Rule 18–0–5	(Required)	The unbounded functions of library *<cstring>* shall not be used.

[Undefined 5.7]

Rationale

The *strcpy*, *strcmp*, *strcat*, *strchr*, *strspn*, *strcspn*, *strpbrk*, *strrchr*, *strstr*, *strtok* and *strlen* functions within the <*cstring*> library can read or write beyond the end of a buffer, resulting in *undefined behaviour*.

Ideally, a safe string handling library should be used.

Example

```
#include <cstring>
void fn ( const char_t * pChar )
{
   char_t array [ 10 ];
   strcpy ( array, pChar );   // Non-compliant
}
```

6.18.2 Implementation properties

Rule 18–2–1 (Required) The macro *offsetof* shall not be used.

Rationale

Use of this macro can lead to *undefined behaviour* when the types of the operands are incompatible, or when bit fields are used.

Example

```
#include <cstddef>
struct A
{
   int32_t i;
};
void f1 ( )
{
   offsetof ( A, i );      // Non-compliant
}
```

6.18.4 Dynamic memory management

Rule 18–4–1 (Required) Dynamic heap memory allocation shall not be used.

Rationale

The use of dynamic memory can lead to out-of-storage run-time failures, which are undesirable.

The built-in *new* and *delete* operators, other than the placement versions, use dynamic heap memory. The functions *calloc*, *malloc*, *realloc* and *free* also use dynamic heap memory.

There is a range of *unspecified*, *undefined* and *implementation-defined behaviour* associated with dynamic memory allocation, as well as a number of other potential pitfalls. Dynamic heap memory allocation may lead to memory leaks, data inconsistency, memory exhaustion, non-deterministic behaviour, etc.

Note that some implementations may use dynamic heap memory allocation to implement other functions (for example, functions in the library *cstring*). If this is the case, then these functions shall also be avoided.

Example

```
void f1 ( )
{
   int32_t * i = new int32_t;       // Non-compliant
   delete i;
}
```

6.18.7 Other runtime support

Rule 18–7–1	**(Required)**	**The signal handling facilities of <*csignal*> shall not be used.**

Rationale

Signal handling contains *implementation-defined* and *undefined behaviour*.

Example

```
#include <csignal>
void my_handler ( int32_t );
void f1 ( )
{
   signal ( 1, my_handler );   // Non-compliant
}
```

6.19 Diagnostics library

6.19.3 Error numbers

Rule 19–3–1	**(Required)**	**The error indicator *errno* shall not be used.**

Rationale

errno is a facility of C++ which should in theory be useful, but which in practice is poorly defined by ISO/IEC 14882:2003 [1]. A non-zero value may or may not indicate that a problem has occurred; therefore *errno* shall not be used.

Even for those functions for which the behaviour of *errno* is well defined, it is preferable to check the values of inputs before calling the function rather than relying on using *errno* to trap errors.

Example

```
#include <cstdlib>
#include <cerrno>
void f1 ( const char_t * str )
{
   errno = 0;                  // Non-compliant
   int32_t i = atoi ( str );
```

```
    if ( 0 != errno )           // Non-compliant
    {
       // handle error case???
    }
}
```

See also

Rule 0–3–2

6.27 Input/output library

6.27.0 General

Rule 27–0–1	(Required)	The stream input/output library <*cstdio*> shall not be used.

Rationale

This includes file and I/O functions *fgetpos*, *fopen*, *ftell*, *gets*, *perror*, *remove*, *rename*, etc.

Streams and file I/O have a large number of *unspecified*, *undefined* and *implementation-defined behaviours* associated with them.

Example

```
#include <cstdio>    // Non-compliant
void fn ( )
{
   char_t array [ 10 ];
   gets ( array );   // Can lead to buffer over-run
}
```

7. References

[1] ISO/IEC 14882:2003, *The C++ Standard Incorporating Technical Corrigendum 1*, International Organization for Standardization, 2003.

[2] MISRA *Development Guidelines for Vehicle Based Software*, ISBN 0-9524156-0-7, Motor Industry Research Association, Nuneaton, November 1994.

[3] MISRA AC INT *Introduction to the MISRA guidelines for the use of automatic code generation in automotive systems*, ISBN 978-1-906400-00-2, MIRA Limited, November 2007.

[4] CRR80, *The Use of Commercial Off-the-Shelf (COTS) Software in Safety Related Applications*, ISBN 0-7176-0984-7, HSE Books.

[5] ISO 9001:2000, *Quality management systems — Requirements*, International Organization for Standardization, 2000.

[6] ISO 90003:2004, *Software engineering — Guidelines for the application of ISO 9001:2000 to computer software*, ISO, 2004.

[7] The TickIT Guide, *Using ISO 9001:2000 for Software Quality Management System Construction, Certification and Continual Improvement*, Issue 5, British Standards Institution, 2001.

[8] Straker D., *C Style: Standards and Guidelines*, ISBN 0–13-116898-3, Prentice Hall 1991.

[9] Fenton N.E. and Pfleeger S.L., *Software Metrics: A Rigorous and Practical Approach*, 2nd Edition, ISBN 0-534-95429-1, PWS, 1998.

[10] MISRA Report 5, *Software Metrics*, Motor Industry Research Association, Nuneaton, February 1995.

[11] MISRA Report 6, *Verification and Validation*, Motor Industry Research Association, Nuneaton, February 1995.

[12] IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, International Electromechanical Commission, in 7 parts published between 1998 and 2000.

[13] Goldberg D., *What Every Computer Scientist Should Know about Floating-Point Arithmetic*, Computing Surveys, March 1991.

[14] ANSI/IEEE Std 754, *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[15] ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Sets (UCS)*, International Organization for Standardization, 2003.

[16] ISO/IEC 9899:1990, *Programming Languages — C*, International Organization for Standardization, 1990.

[17] Hill M.G. and Whiting E.V, *An Investigation of the Unpredictable Features of the C++ Language*, QINETIQ/KI/TIM/TR043014, QinetiQ, May 2004.

[18] *High-Integrity C++ Coding Standard Manual Version 2.2*, The Programming Research Group, May 2004.

7. References (continued)

[19] *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Document Number 2RDU00001 Rev C, Lockheed Martin, December 2005.

[20] Henricson M., Nyquist E., I*ndustrial Strength C++*, ISBN 0-13-120965-5, Prentice Hall, 1997.

[21] Sutter H., *Exceptional C++*, ISBN 0-201-61562-2, Addison-Wesley, 1999.

[22] Koenig A., C *Traps and Pitfalls*, ISBN 0-201-17928-8, Addison-Wesley, 1988.

[23] Hatton L., *Safer C*, ISBN 0-07-707640-0, McGraw-Hill, 1994.

[24] Dewhurst S., *C++ Gotchas*, ISBN 0-321-12518-5, Addison-Wesley, 2003.

[25] Meyers S., *More Effective C++*, ISBN 0-201-63371-X, Addison-Wesley, 1996.

[26] Meyers S., *Effective C++*, ISBN 0-321-33487-6 (Third Edition), Addison-Wesley, 2005.

Appendix A

Appendix A: Summary of rules

Unnecessary constructs

Rule 0–1–1	(Required)	A *project* shall not contain *unreachable code*.
Rule 0–1–2	(Required)	A *project* shall not contain *infeasible paths*.
Rule 0–1–3	(Required)	A *project* shall not contain *unused* variables.
Rule 0–1–4	(Required)	A *project* shall not contain non-volatile *POD* variables having only one *use*.
Rule 0–1–5	(Required)	A *project* shall not contain *unused* type declarations.
Rule 0–1–6	(Required)	A *project* shall not contain instances of non-volatile variables being given values that are never subsequently *used*.
Rule 0–1–7	(Required)	The value returned by a function having a non-*void* return type that is not an overloaded operator shall always be *used*.
Rule 0–1–8	(Required)	All functions with *void* return type shall have external side effect(s).
Rule 0–1–9	(Required)	There shall be no *dead code*.
Rule 0–1–10	(Required)	Every defined function shall be called at least once.
Rule 0–1–11	(Required)	There shall be no *unused* parameters (named or unnamed) in non-virtual functions.
Rule 0–1–12	(Required)	There shall be no *unused* parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Storage

Rule 0–2–1	(Required)	An object shall not be assigned to an overlapping object.

Runtime failures

Rule 0–3–1	(Document)	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.
Rule 0–3–2	(Required)	If a function generates error information, then that error information shall be tested.

Arithmetic

Rule 0–4–1	(Document)	Use of scaled-integer or fixed-point arithmetic shall be documented.
Rule 0–4–2	(Document)	Use of floating-point arithmetic shall be documented.

Rule 0–4–3	(Document)	Floating-point implementations shall comply with a defined floating-point standard.

Language

Rule 1–0–1	(Required)	All code shall conform to ISO/IEC 14882:2003 “The C++ Standard Incorporating Technical Corrigendum 1”.
Rule 1–0–2	(Document)	Multiple compilers shall only be used if they have a common, defined interface.
Rule 1–0–3	(Document)	The implementation of integer division in the chosen compiler shall be determined and documented.

Character sets

Rule 2–2–1	(Document)	The character set and the corresponding encoding shall be documented.

Trigraph sequences

Rule 2–3–1	(Required)	*Trigraphs* shall not be used.

Alternative tokens

Rule 2–5–1	(Advisory)	*Digraphs* should not be used.

Comments

Rule 2–7–1	(Required)	The character sequence `/*` shall not be used within a C-style comment.
Rule 2–7–2	(Required)	Sections of code shall not be “commented out” using C-style comments.
Rule 2–7–3	(Advisory)	Sections of code should not be “commented out” using C++ comments.

Identifiers

Rule 2–10–1	(Required)	Different identifiers shall be typographically unambiguous.
Rule 2–10–2	(Required)	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
Rule 2–10–3	(Required)	A *typedef* name (including qualification, if any) shall be a *unique* identifier.
Rule 2–10–4	(Required)	A *class*, *union* or *enum* name (including qualification, if any) shall be a *unique* identifier.
Rule 2–10–5	(Advisory)	The identifier name of a non-member object or function with static storage duration should not be reused.
Rule 2–10–6	(Required)	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Literals

Rule 2–13–1 (Required) Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Rule 2–13–2 (Required) Octal constants (other than zero) and octal escape sequences (other than "`\0`") shall not be used.

Rule 2–13–3 (Required) A "*U*" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Rule 2–13–4 (Required) Literal suffixes shall be upper case.

Rule 2–13–5 (Required) Narrow and wide string literals shall not be concatenated.

Declarations and definitions

Rule 3–1–1 (Required) It shall be possible to include any *header file* in multiple translation units without violating the *One Definition Rule*.

Rule 3–1–2 (Required) Functions shall not be declared at block scope.

Rule 3–1–3 (Required) When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

One Definition Rule

Rule 3–2–1 (Required) All declarations of an object or function shall have *compatible types*.

Rule 3–2–2 (Required) The *One Definition Rule* shall not be violated.

Rule 3–2–3 (Required) A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Rule 3–2–4 (Required) An identifier with external linkage shall have exactly one definition.

Declarative regions and scope

Rule 3–3–1 (Required) Objects or functions with external linkage shall be declared in a *header file*.

Rule 3–3–2 (Required) If a function has internal linkage then all re-declarations shall include the *static* storage class specifier.

Name lookup

Rule 3–4–1 (Required) An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Types

Rule 3–9–1 (Required) The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Rule 3–9–2	(Advisory)	*typedefs* that indicate size and signedness should be used in place of the basic numerical types.
Rule 3–9–3	(Required)	The underlying bit representations of floating-point values shall not be used.

Integral promotions

Rule 4–5–1	(Required)	Expressions with type *bool* shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.
Rule 4–5–2	(Required)	Expressions with type *enum* shall not be used as operands to built-in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.
Rule 4–5–3	(Required)	Expressions with type (plain) *char* and *wchar_t* shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Pointer conversions

Rule 4–10–1	(Required)	*NULL* shall not be used as an integer value.
Rule 4–10–2	(Required)	Literal zero (0) shall not be used as the *null-pointer-constant*.

Expressions

Rule 5–0–1	(Required)	The value of an expression shall be the same under any order of evaluation that the standard permits.
Rule 5–0–2	(Advisory)	Limited dependence should be placed on C++ operator precedence rules in expressions.
Rule 5–0–3	(Required)	A *cvalue* expression shall not be implicitly converted to a different *underlying type*.
Rule 5–0–4	(Required)	An implicit integral conversion shall not change the signedness of the *underlying type*.
Rule 5–0–5	(Required)	There shall be no implicit *floating-integral* conversions.
Rule 5–0–6	(Required)	An implicit integral or floating-point conversion shall not reduce the size of the *underlying type*.
Rule 5–0–7	(Required)	There shall be no explicit *floating-integral* conversions of a *cvalue* expression.
Rule 5–0–8	(Required)	An explicit integral or floating-point conversion shall not increase the size of the *underlying type* of a *cvalue* expression.
Rule 5–0–9	(Required)	An explicit integral conversion shall not change the signedness of the *underlying type* of a *cvalue* expression.

Rule 5–0–10	(Required)	If the bitwise operators ~ and << are applied to an operand with an *underlying type* of *unsigned char* or *unsigned short*, the result shall be immediately cast to the *underlying type* of the operand.
Rule 5–0–11	(Required)	The plain *char* type shall only be used for the storage and use of character values.
Rule 5–0–12	(Required)	*signed char* and *unsigned char* type shall only be used for the storage and use of numeric values.
Rule 5–0–13	(Required)	The *condition* of an *if-statement* and the *condition* of an *iteration-statement* shall have type *bool*.
Rule 5–0–14	(Required)	The first operand of a *conditional-operator* shall have type *bool*.
Rule 5–0–15	(Required)	Array indexing shall be the only form of pointer arithmetic.
Rule 5–0–16	(Required)	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.
Rule 5–0–17	(Required)	Subtraction between pointers shall only be applied to pointers that address elements of the same array.
Rule 5–0–18	(Required)	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
Rule 5–0–19	(Required)	The declaration of objects shall contain no more than two levels of pointer indirection.
Rule 5–0–20	(Required)	Non-constant operands to a binary bitwise operator shall have the same *underlying type*.
Rule 5–0–21	(Required)	Bitwise operators shall only be applied to operands of unsigned *underlying type*.

Postfix expressions

Rule 5–2–1	(Required)	Each operand of a logical && or || shall be a *postfix-expression*.
Rule 5–2–2	(Required)	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of *dynamic_cast*.
Rule 5–2–3	(Advisory)	Casts from a base class to a derived class should not be performed on polymorphic types.
Rule 5–2–4	(Required)	C-style casts (other than *void* casts) and functional notation casts (other than explicit constructor calls) shall not be used.
Rule 5–2–5	(Required)	A cast shall not remove any *const* or *volatile* qualification from the type of a pointer or reference.
Rule 5–2–6	(Required)	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
Rule 5–2–7	(Required)	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

Rule 5–2–8	(Required)	An object with integer type or pointer to *void* type shall not be converted to an object with pointer type.
Rule 5–2–9	(Advisory)	A cast should not convert a pointer type to an integral type.
Rule 5–2–10	(Advisory)	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.
Rule 5–2–11	(Required)	The comma operator, && operator and the || operator shall not be overloaded.
Rule 5–2–12	(Required)	An identifier with array type passed as a function argument shall not decay to a pointer.

Unary expressions

Rule 5–3–1	(Required)	Each operand of the ! operator, the logical && or the logical || operators shall have type *bool*.
Rule 5–3–2	(Required)	The unary minus operator shall not be applied to an expression whose *underlying type* is unsigned.
Rule 5–3–3	(Required)	The unary & operator shall not be overloaded.
Rule 5–3–4	(Required)	Evaluation of the operand to the *sizeof* operator shall not contain side effects.

Shift operators

Rule 5–8–1	(Required)	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the *underlying type* of the left hand operand.

Logical AND operator

Rule 5–14–1	(Required)	The right hand operand of a logical && or || operator shall not contain side effects.

Assignment operators

Rule 5–17–1	(Required)	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

Comma operator

Rule 5–18–1	(Required)	The comma operator shall not be used.

Constant expressions

Rule 5–19–1	(Advisory)	Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Expression statement

Rule 6–2–1	(Required)	Assignment operators shall not be used in sub-expressions.

Rule 6–2–2	(Required)	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
Rule 6–2–3	(Required)	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Compound statement

Rule 6–3–1	(Required)	The statement forming the body of a *switch, while, do ... while* or *for* statement shall be a compound statement.

Selection statements

Rule 6–4–1	(Required)	An *if (condition)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement.
Rule 6–4–2	(Required)	All *if … else if* constructs shall be terminated with an *else* clause.
Rule 6–4–3	(Required)	A *switch* statement shall be a *well-formed switch statement*.
Rule 6–4–4	(Required)	A *switch-label* shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement.
Rule 6–4–5	(Required)	An unconditional *throw* or *break* statement shall terminate every non-empty *switch-clause*.
Rule 6–4–6	(Required)	The final clause of a *switch* statement shall be the *default-clause*.
Rule 6–4–7	(Required)	The *condition* of a *switch* statement shall not have *bool* type.
Rule 6–4–8	(Required)	Every *switch* statement shall have at least one *case-clause*.

Iteration statements

Rule 6–5–1	(Required)	A *for* loop shall contain a single *loop-counter* which shall not have floating type.
Rule 6–5–2	(Required)	If *loop-counter* is not modified by `--` or `++`, then, within *condition*, the *loop-counter* shall only be used as an operand to `<=`, `<`, `>` or `>=`.
Rule 6–5–3	(Required)	The *loop-counter* shall not be modified within *condition* or *statement*.
Rule 6–5–4	(Required)	The *loop-counter* shall be modified by one of: `--`, `++`, `-=n`, or `+=n`; where `n` remains constant for the duration of the loop.
Rule 6–5–5	(Required)	A *loop-control-variable* other than the *loop-counter* shall not be modified within *condition* or *expression*.
Rule 6–5–6	(Required)	A *loop-control-variable* other than the *loop-counter* which is modified in *statement* shall have type *bool*.

Jump statements

Rule 6–6–1	(Required)	Any label referenced by a *goto* statement shall be declared in the same block, or in a block enclosing the *goto* statement.

Rule 6–6–2	(Required)	The *goto* statement shall jump to a label declared later in the same function body.
Rule 6–6–3	(Required)	The *continue* statement shall only be used within a *well-formed for loop*.
Rule 6–6–4	(Required)	For any iteration statement there shall be no more than one *break* or *goto* statement used for loop termination.
Rule 6–6–5	(Required)	A function shall have a single point of exit at the end of the function.

Specifiers

Rule 7–1–1	(Required)	A variable which is not modified shall be *const* qualified.
Rule 7–1–2	(Required)	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Enumeration declarations

Rule 7–2–1	(Required)	An expression with *enum underlying type* shall only have values corresponding to the enumerators of the enumeration.

Namespaces

Rule 7–3–1	(Required)	The global namespace shall only contain *main*, namespace declarations and *extern "C"* declarations.
Rule 7–3–2	(Required)	The identifier *main* shall not be used for a function other than the global function *main*.
Rule 7–3–3	(Required)	There shall be no unnamed namespaces in *header files*.
Rule 7–3–4	(Required)	*using-directives* shall not be used.
Rule 7–3–5	(Required)	Multiple declarations for an identifier in the same namespace shall not straddle a *using-declaration* for that identifier.
Rule 7–3–6	(Required)	*using-directives* and *using-declarations* (excluding class scope or function scope *using-declarations*) shall not be used in *header files*.

The *asm* declaration

Rule 7–4–1	(Document)	All usage of assembler shall be documented.
Rule 7–4–2	(Required)	Assembler instructions shall only be introduced using the *asm* declaration.
Rule 7–4–3	(Required)	Assembly language shall be encapsulated and isolated.

Linkage specifications

Rule 7–5–1	(Required)	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Rule 7–5–2	(Required)	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
Rule 7–5–3	(Required)	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.
Rule 7–5–4	(Advisory)	Functions should not call themselves, either directly or indirectly.

Declarators — General

Rule 8–0–1	(Required)	An *init-declarator-list* or a *member-declarator-list* shall consist of a single *init-declarator* or *member-declarator* respectively.

Meaning of declarators

Rule 8–3–1	(Required)	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Function definitions

Rule 8–4–1	(Required)	Functions shall not be defined using the ellipsis notation.
Rule 8–4–2	(Required)	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.
Rule 8–4–3	(Required)	All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression.
Rule 8–4–4	(Required)	A function identifier shall either be used to call the function or it shall be preceded by &.

Declarators — Initializers

Rule 8–5–1	(Required)	All variables shall have a defined value before they are used.
Rule 8–5–2	(Required)	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
Rule 8–5–3	(Required)	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Member functions

Rule 9–3–1	(Required)	const member functions shall not return non-const pointers or references to *class-data*.
Rule 9–3–2	(Required)	Member functions shall not return non-const *handles* to *class-data*.
Rule 9–3–3	(Required)	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Unions

Rule 9–5–1	(Required)	Unions shall not be used.

Bit-fields

Rule 9–6–1	(Document)	When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.
Rule 9–6–2	(Required)	Bit-fields shall be either *bool* type or an explicitly *unsigned* or *signed* integral type.
Rule 9–6–3	(Required)	Bit-fields shall not have *enum* type.
Rule 9–6–4	(Required)	Named bit-fields with *signed* integer type shall have a length of more than one bit.

Multiple base classes

Rule 10–1–1	(Advisory)	Classes should not be derived from virtual bases.
Rule 10–1–2	(Required)	A base class shall only be declared virtual if it is used in a diamond hierarchy.
Rule 10–1–3	(Required)	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Member name lookup

Rule 10–2–1	(Advisory)	All accessible entity names within a multiple inheritance hierarchy should be *unique*.

Virtual functions

Rule 10–3–1	(Required)	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
Rule 10–3–2	(Required)	Each overriding virtual function shall be declared with the *virtual* keyword.
Rule 10–3–3	(Required)	A virtual function shall only be overridden by a *pure virtual function* if it is itself declared as *pure virtual*.

Member access control — General

Rule 11–0–1	(Required)	Member data in non-*POD* class types shall be private.

Constructors

Rule 12–1–1	(Required)	An object's dynamic type shall not be used from the body of its constructor or destructor.
Rule 12–1–2	(Advisory)	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.
Rule 12–1–3	(Required)	All constructors that are callable with a single argument of fundamental type shall be declared *explicit*.

Copying class objects

Rule 12–8–1 (Required) A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.

Rule 12–8–2 (Required) The copy assignment operator shall be declared *protected* or *private* in an abstract class.

Template declarations

Rule 14–5–1 (Required) A non-member *generic function* shall only be declared in a namespace that is not an *associated namespace*.

Rule 14–5–2 (Required) A *copy constructor* shall be declared when there is a template constructor with a single parameter that is a *generic parameter*.

Rule 14–5–3 (Required) A *copy assignment operator* shall be declared when there is a template assignment operator with a parameter that is a *generic parameter.*

Name resolution

Rule 14–6–1 (Required) In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a *qualified-id* or `this->`

Rule 14–6–2 (Required) The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

Template instantiation and specialization

Rule 14–7–1 (Required) All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.

Rule 14–7–2 (Required) For any given template specialization, an explicit instantiation of the template with the *template-arguments* used in the specialization shall not render the program ill-formed.

Rule 14–7–3 (Required) All partial and explicit specializations for a template shall be declared in the same file as the declaration of their *primary template*.

Function template specialization

Rule 14–8–1 (Required) Overloaded function templates shall not be explicitly specialized.

Rule 14–8–2 (Advisory) The viable *function set* for a function call should either contain no function specializations, or only contain function specializations.

Exception handling — General

Rule 15–0–1 (Document) Exceptions shall only be used for error handling.

Rule 15–0–2 (Advisory) An exception object should not have pointer type.

Rule 15–0–3 (Required) Control shall not be transferred into a *try* or *catch* block using a *goto* or a *switch* statement.

Throwing an exception

Rule 15–1–1 (Required) The *assignment-expression* of a *throw* statement shall not itself cause an exception to be thrown.

Rule 15–1–2 (Required) *NULL* shall not be thrown explicitly.

Rule 15–1–3 (Required) An empty *throw* (`throw;`) shall only be used in the *compound-statement* of a *catch* handler.

Handling an exception

Rule 15–3–1 (Required) Exceptions shall be raised only after start-up and before termination of the program.

Rule 15–3–2 (Advisory) There should be at least one exception handler to catch all otherwise unhandled exceptions

Rule 15–3–3 (Required) Handlers of a *function-try-block* implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Rule 15–3–4 (Required) Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Rule 15–3–5 (Required) A class type exception shall always be caught by reference.

Rule 15–3–6 (Required) Where multiple handlers are provided in a single *try-catch* statement or *function-try-block* for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Rule 15–3–7 (Required) Where multiple handlers are provided in a single *try-catch* statement or *function-try-block*, any ellipsis (catch-all) handler shall occur last.

Exception specifications

Rule 15–4–1 (Required) If a function is declared with an *exception-specification*, then all declarations of the same function (in other translation units) shall be declared with the same set of *type-ids*.

Exception handling — Special functions

Rule 15–5–1 (Required) A class destructor shall not exit with an exception.

Rule 15–5–2 (Required) Where a function's declaration includes an *exception-specification*, the function shall only be capable of throwing exceptions of the indicated type(s).

Rule 15–5–3 (Required) The *terminate()* function shall not be called implicitly.

Preprocessing directives — General

Rule 16–0–1 (Required) *#include* directives in a file shall only be preceded by other preprocessor directives or comments.

Rule 16–0–2	(Required)	Macros shall only be *#define*'d or *#undef*'d in the global namespace.
Rule 16–0–3	(Required)	*#undef* shall not be used.
Rule 16–0–4	(Required)	Function-like macros shall not be defined.
Rule 16–0–5	(Required)	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
Rule 16–0–6	(Required)	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
Rule 16–0–7	(Required)	Undefined macro identifiers shall not be used in *#if* or *#elif* preprocessor directives, except as operands to the *defined* operator.
Rule 16–0–8	(Required)	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Conditional inclusion

Rule 16–1–1	(Required)	The *defined* preprocessor operator shall only be used in one of the two standard forms.
Rule 16–1–2	(Required)	All *#else*, *#elif* and *#endif* preprocessor directives shall reside in the same file as the *#if* or *#ifdef* directive to which they are related.

Source file inclusion

Rule 16–2–1	(Required)	The pre-processor shall only be used for file inclusion and *include guards*.
Rule 16–2–2	(Required)	C++ macros shall only be used for: *include guards*, type qualifiers, or storage class specifiers.
Rule 16–2–3	(Required)	*Include guards* shall be provided.
Rule 16–2–4	(Required)	The `'`, `"`, `/*` or `//` characters shall not occur in a *header file* name.
Rule 16–2–5	(Advisory)	The `\` character should not occur in a *header file* name.
Rule 16–2–6	(Required)	The *#include* directive shall be followed by either a <*filename*> or *"filename"* sequence.

Macro replacement

Rule 16–3–1	(Required)	There shall be at most one occurrence of the # or ## operators in a single macro definition.
Rule 16–3–2	(Advisory)	The # and ## operators should not be used.

Pragma directive

Rule 16–6–1	(Document)	All uses of the *#pragma* directive shall be documented.

Library introduction — General

Rule 17–0–1 (Required) Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Rule 17–0–2 (Required) The names of standard library macros and objects shall not be reused.

Rule 17–0–3 (Required) The names of standard library functions shall not be overridden.

Rule 17–0–4 (Document) All library code shall conform to MISRA C++.

Rule 17–0–5 (Required) The *setjmp* macro and the *longjmp* function shall not be used.

Language support library — General

Rule 18–0–1 (Required) The C library shall not be used.

Rule 18–0–2 (Required) The library functions *atof*, *atoi* and *atol* from library <*cstdlib*> shall not be used.

Rule 18–0–3 (Required) The library functions *abort*, *exit*, *getenv* and *system* from library <*cstdlib*> shall not be used.

Rule 18–0–4 (Required) The time handling functions of library <*ctime*> shall not be used.

Rule 18–0–5 (Required) The unbounded functions of library <*cstring*> shall not be used.

Language support library — Implementation properties

Rule 18–2–1 (Required) The macro *offsetof* shall not be used.

Language support library — Dynamic memory management

Rule 18–4–1 (Required) Dynamic heap memory allocation shall not be used.

Language support library — Other runtime support

Rule 18–7–1 (Required) The signal handling facilities of <*csignal*> shall not be used.

Diagnostics library — Error numbers

Rule 19–3–1 (Required) The error indicator *errno* shall not be used.

Input/output library — General

Rule 27–0–1 (Required) The stream input/output library <*cstdio*> shall not be used.

Appendix B

Appendix B: C++ vulnerabilities

Unlike the C Language Reference Manual [16], the C++ *LRM* (ISO/IEC 14882:2003 [1]) does not include a collated list of the features of the language that are "unspecified", "undefined", etc. This annex provides such a list.

The extraction of this information from the *LRM* was funded by the Analysis, Experimentation and Simulation Domain of the UK Ministry of Defence Scientific Research Programme, and is quoted here with permission.

Each entry in the list contains the following information:

- MISRA Id: an identifier unique to this document, to allow easy reference to particular issues.
- ISO Reference: a reference into the *LRM*, in the form X.Y(Z) being paragraph Z of subsection Y of section X, etc..
- Description: a brief summary of the unspecified etc. behaviour described in the referenced paragraph.
- Category: the sort of behaviour described in the referenced paragraph (see below).
- MISRA Guidance: which Rule(s) in this document address the described issue.
- Source Id: unique identifier used in the document where this table was originally published [17], to allow cross-referencing.

The categories are as follows:

- Unspecified: A situation where the implementation will have to make some "sensible" choice, but that choice is not predictable by the programmer, e.g. the order in which sub-expressions are evaluated in an expression.
- Undefined: A situation where the *LRM* can give no indication of what behaviour to expect from a program. This behaviour may result in catastrophic failure (a "crash") or continued execution with some arbitrary data.
- Implementation: A situation where the implementation will have to make some "sensible" choice and where that choice has to be documented and be available to the programmer, e.g. the size of integers.
- Indeterminate: A sub-category of undefined behaviour, where the *LRM* says "if condition, the behaviour is ..." but does not say what happens if the condition is not true.
- Behaviour that requires no diagnostics: A situation where the *LRM* permits behaviour that may be unexpected or at odds with previously defined principles, but explicitly the *LRM* does not require the programmer to be warned.

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
1	2.1(1)	The mapping of physical source file characters.	Implementation		3.01
2	2.1(2)	A character sequence that matches a universal-character-name is produced due to the splicing of physical source lines in the translation process.	Undefined		2.01
3	2.1(2)	A non-empty source file does not end in a new line character, or ends in a new line character immediately preceded by a backslash character.	Undefined		2.02
4	2.1(2)	Use of an identifier reserved for C++ implementations and standard libraries.	Behaviour that requires no diagnostics		5.02
5	2.1(3)	Whether each non-empty sequence of white-space characters other than new line is retained or replaced by one space character.	Implementation		3.02
6	2.1(4)	A character sequence that matches a universal-character-name is produced due to token concatenation.	Undefined		2.03
7	2.1(8)	Whether the source of the translational units containing the definitions of the templates for the requisite instantiations is required to be available.	Implementation		3.03
8	2.2(3)	The values of the members of the execution character sets.	Implementation		3.04
9	2.4(2)	An unmatched ' or a " character is encountered on a logical source line during tokenization.	Undefined		2.04
10	2.7(1)	A // comment contains a form feed or vertical-tab character and does not only have white space characters between it and the new-line that terminates the comment.	Behaviour that requires no diagnostics		5.01
11	2.8(1)	The mapping of the sequences in both forms of *header-names*. See 16.2(2)	Implementation		3.05
12	2.8(2)	The characters ', \, ", /*, or // are encountered between the < and > delimiters or the characters ', \, /*, or // are encountered between the " delimiters in the two forms of a header name preprocessing token.	Undefined	Rule 16–2–4 Rule 16–2–5	2.05
13	2.13.1(2)	An integer literal cannot be represented by any of the allowed types.	Undefined		2.06
14	2.13.2(1)	The value of a multi-character literal.	Implementation		3.06
15	2.13.2(2)	The value of a wide-character literal containing multiple *c-chars*.	Implementation		3.07
16	2.13.2(3)	The character following a backslash does not give a valid escape sequence.	Undefined	Rule 2–13–1	2.07

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
17	2.13.2(4)	The value of a character literal that falls outside of the implementation defined range for *char* or *w_char*.	Implementation		3.08
18	2.13.2(5)	The encoding of a universal-character-name where the execution character set has no encoding for the character named.	Implementation		3.09
19	2.13.3(1)	The actual value used for a floating literal whose value is not in the range of representable values for its type.	Implementation		3.10
20	2.13.4(2)	An attempt is made to modify a string literal.	Undefined		2.08
21	2.13.4(2)	Whether all string literals are distinct (stored in non-overlapping objects).	Implementation		3.11
22	2.13.4(3)	A narrow string literal token is adjacent to a wide string literal token.	Undefined	Rule 2–13–5	2.09
23	3.1(15)	A program attempting to access the stored value of an object through an lvalue of other than one of the types specified.	Undefined		2.22
24	3.2(3)	A program that does not contain exactly one definition for every non-inline function or object that is used in that program.	Behaviour that requires no diagnostics	Rule 3–2–1 Rule 3–2–4	5.03
25	3.2(5)	The behaviour of a program if two definitions in separate translation units do not satisfy the One Definition Rule.	Undefined	Rule 3–2–1 Rule 3–2–4	2.10
26	3.3.1(1)	The value used when a variable is used to initialize itself, e.g. *int x = x*;	Indeterminate		4.01
27	3.3.6(1) Item 2	A name N used in a class S does not refer to the same declaration in its context and when re-evaluated in the completed scope of S.	Behaviour that requires no diagnostics		5.04
28	3.3.6(1) Item 3	If reordering member declarations in a class yields an alternative valid program under certain conditions.	Behaviour that requires no diagnostics		5.05
29	3.5(10)	If a given object or function can be referred to by values of different type (after all types adjustments)	Behaviour that requires no diagnostics		5.06
30	3.6.1(1)	Whether a program in a freestanding environment is required to define a *main* function.	Implementation		3.12
31	3.6.1(1)	Start-up and termination in a freestanding environment.	Implementation		3.13
32	3.6.1(2)	The type of the *main* function, though its return type must be int.	Implementation	Rule 7–3–2	3.14
33	3.6.1(3)	The linkage of *main*.	Implementation	Rule 7–3–2	3.15
34	3.6.1(4)	The library function *exit* is called to end a program during the destruction of an object with static storage duration.	Undefined		2.11

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
35	3.6.2(2)	Whether an object is fully, or merely zero-initialized when an object refers to another object of namespace scope with static storage duration potentially requiring dynamic initialization and defined later in the same translational unit.	Unspecified		1.01
36	3.6.2(3)	Whether the dynamic initialization of an object of namespace scope is done before the first statement of main.	Implementation		3.16
37	3.6.3(2)	A function contains a local object of static storage duration that has been destroyed and the function is called during the destruction of an object with static storage duration and the flow of control passes through the definition of the previously destroyed object.	Undefined		2.12
38	3.7.3.1(2)	The order, contiguity and initial value of storage allocated by the allocation functions.	Unspecified		1.02
39	3.7.3.1(2)	The results of dereferencing a pointer returned as a request for zero size space in a call to an allocation function.	Undefined		2.13
40	3.7.3.2(4)	Attempt to use a pointer to a deleted object.	Undefined		2.14
41	3.8(4)	The side effects of a non-trivial destructor of an object of class type whose lifetime has ended, but whose destructor has not been called explicitly.	Undefined		2.15
42	3.8(5)	An object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*.	Undefined		2.16
43	3.8(5)	Series of uses of a pointer to a non-POD class type between object storage allocation and the start of object lifetime, and the end of object lifetime and storage de-allocation.	Undefined		2.17
44	3.8(6)	An lvalue-to-rvalue conversion is applied to an lvalue that refers to an object whose lifetime has not yet started but whose storage has been allocated, or whose lifetime has ended but whose storage has not been reused or released.	Undefined		2.18
45	3.8(6)	Series of uses of an lvalue that refers to a non-POD class type between object storage allocation and the start of object lifetime, and the end of object lifetime and storage deallocation.	Undefined		2.19

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
46	3.8(8)	A program ends the lifetime of an object of type T with static or automatic storage duration, T has a non-trivial destructor and an object of a different type occupies the storage location when the implicit destructor call takes place.	Undefined		2.20
47	3.8(9)	A new object is created at the storage location that a const object with static or automatic storage duration occupies or, at the storage location that such a const object used to occupy before its lifetime ended.	Undefined		2.21
48	3.9(4)	For POD types, the set of values of which the value representation (a set of bits in the object representation that determines a *value*) is one discrete element.	Implementation	Rule 9–5–1	3.17
49	3.9(5)	The packing needed between sub-objects to meet alignment requirements	Implementation	Rule 9–5–1	3.18
50	3.9.1(1)	Whether *char* is equivalent to *unsigned char* or *signed char*.	Implementation	Rule 3–9–2 Rule 5–0–11 Rule 5–0–12	3.19
51	3.9.1(2)	Size of *int*.	Implementation		3.20
52	3.9.1(5)	Type of *wchar_t*.	Implementation	Rule 3–9–2	3.21
53	3.9.1(8)	The value representation of floating-point types.	Implementation	Rule 3–9–3	3.22
54	3.9.2(3)	The value representation of pointer types.	Implementation		3.23
55	4.1(1)	An lvalue, which does not refer to an object of type T or is uninitialized, is used where an rvalue of type T is expected.	Undefined		2.23
56	4.7(3)	The value of a signed integer type due to the conversion from either an integer or an enumeration type when the value cannot be represented in the destination type.	Implementation		3.24
57	4.8(1)	A floating-point conversion produces a result that cannot be represented in the space provided.	Undefined		2.24
58	4.8(1)	The value resulting from converting a value of a floating point type to another floating point type that cannot exactly represent the original value.	Implementation		3.25
59	4.9(1)	A floating-integral conversion produces a result that cannot be represented in the space provided.	Undefined	Rule 5–0–5 Rule 5–0–6	2.25
60	4.9(2)	The choice of either the next higher or lower representable value when an rvalue of an integer or enumeration type is converted to an rvalue of a floating-point type but exact conversion is not possible.	Implementation		3.26

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
61	5(4)	The order of evaluation of operands of individual operators and sub-expressions of individual expressions, and the order in which side effects take place.	Unspecified	Rule 5–0–1	1.03
62	5(4)	An object is modified more than once or is modified and accessed other than to determine the new value, between two sequence points.	Undefined	Rule 5–0–1	2.26
63	5(5)	An arithmetic operation is invalid (such as division or modulus by zero) or produces a result that cannot be represented in the space provided (such as overflow or underflow).	Undefined		2.27
64	5.1(2)	Pointers are compared using an equality operator and either is a pointer to a virtual member function.	Unspecified		1.14
65	5.2.2(1)	A function is called through an expression whose function type has a language linkage that is different from the language linkage of the function type of the called function's definition.	Undefined		2.28
66	5.2.2(7)	An argument with no parameter, after standard conversions, has a non-POD class type.	Undefined	Rule 8–4–1	2.29
67	5.2.2(8)	The order of evaluation of arguments in a function call and the order of evaluation of the postfix expression and the argument expression list.	Unspecified		1.04
68	5.2.8(1)	Whether or not the destructor is called for the *type_info* object at the end of the program.	Unspecified		1.05
69	5.2.8(1)	The class (*name*) derived from *std::type_info* of an lvalue of dynamic type constname, that is the result of a *typeid* expression.	Implementation		3.27
70	5.2.9(5)	A *static_cast* is used to cast an lvalue of class type to a non-derived class.	Undefined	Rule 5–2–2	2.30
71	5.2.9(7)	An integer type is explicitly converted to an enumeration type but the integral value is not within the range of the enumeration values	Unspecified		1.06
72	5.2.9(8)	A *static_cast* is used to cast a pointer of class type to a pointer from a non-derived class.	Undefined	Rule 5–2–2	2.31
73	5.2.9(9)	A *static_cast* is used to cast a pointer to a class member to a pointer to a member of a non-derived class	Undefined		2.32
74	5.2.10(3)	The mapping performed by *reinterpret_cast*.	Implementation		3.28

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
75	5.2.10(4)	The mapping function used to explicitly converting a pointer to any integral type large enough to hold it.	Implementation	Rule 5–2–9	3.29
76	5.2.10(5)	Mappings between pointers and integers other than when a value of integral or enumeration type is explicitly converted into a pointer or when a pointer is converted to an integer of sufficient size and back to the same pointer type.	Implementation	Rule 5–2–9	3.30
77	5.2.10(6)	A pointer to a function is explicitly converted to a function of a different type using *reinterpret_cast*.	Unspecified	Rule 5–2–6	1.07
78	5.2.10(6)	A pointer to a function is converted by *reinterpret_cast* to point to a function of a different type and used to call a function of a type not compatible with the original type.	Undefined	Rule 5–2–6	2.33
79	5.2.10(7)	A pointer to an object is explicitly converted to a pointer to an object of a different type using *reinterpret_cast*.	Unspecified	Rule 5–2–7 Rule 5–2–8	1.08
80	5.2.10(9)	A pointer to member of some type is explicitly converted to a pointer to another member of another type using *reinterpret_cast*.	Unspecified		1.09
81	5.2.11(12)	The use of values produced from conversions between pointers and functions, pointers and member functions and in particular a pointer to a const member function to a pointer to a non-const member function.	Undefined		2.35
82	5.2.11(7)	Depending on the type of object, a write operation through the pointer, lvalue or pointer to data member resulting from a const_char that casts away a const-qualifier may produce undefined behaviour.	Undefined		2.34
83	5.3.1(4)	The address of an object with incomplete type, whose complete type declares *operator&()* as a member function.	Undefined	Rule 5–3–3	2.36
84	5.3.3(1)	The result of *sizeof* applied to any fundamental type (other than *char*, *signed char* and *unsigned char*), in particular *sizeof(bool)* and *sizeof(wchar_t)*.	Implementation		3.31
85	5.3.4(15)	The value of a POD object created by a new-expression when a new-initializer is omitted.	Indeterminate		4.02
86	5.3.4(21)	The order of evaluation of the allocation function and its arguments.	Unspecified		1.10
87	5.3.4(21)	The evaluation of arguments if the allocation function returns null or exits using an exception.	Unspecified		1.11

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
88	5.3.4(6)	The first array dimension applied to a *new* operator is negative.	Undefined		2.37
89	5.3.5(2)	The behaviour of the *delete* operator on a pointer to a non-array object or a pointer to a sub-object representing the base class of such an object that was not obtained from a *new* operator.	Undefined		2.38
90	5.3.5(2)	The value of the operand of delete is not the pointer value that resulted from a previous array new-expression when deleting an array.	Undefined		2.39
91	5.3.5(3)	When deleting an object and the static type of the operand is different from its dynamic type and either the static type is not a base class of the operand's dynamic type, or the static type does not have a virtual destructor.	Undefined		2.40
92	5.3.5(3)	The dynamic type of the object to be deleted differs from its static type when deleting an array.	Undefined		2.41
93	5.3.5(4)	The value of a pointer that refers to deallocated storage	Indeterminate		4.03
94	5.3.5(5)	The object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or deallocation function.	Undefined		2.42
95	5.4(6)	Whether the *static_cast* or *reinterpret_cast* interpretation is used if either the operand or destination type of the cast is a pointer to incomplete class type.	Unspecified		1.12
96	5.5(4)	In a pointer-to-member operation the dynamic type of an object does not contain the member to which the pointer refers.	Undefined		2.43
97	5.5(6)	The second operand of an ->* expression is the null pointer to a member value.	Undefined		2.44
98	5.6(4)	The second operand of the / or % operators is zero.	Undefined		2.45
99	5.6(4)	The sign of the remainder using the binary % operator unless both operands are non-negative.	Implementation	Rule 1–0–3	3.32
100	5.7(5)	A pointer that does not behave like a pointer to an element of an array object is added to or subtracted from.	Undefined	Rule 5–0–16	2.46
101	5.7(5)	The resultant pointer from an addition or subtraction to a pointer to an element of an array which does not point within the array (or one beyond).	Undefined	Rule 5–0–16	2.47
102	5.7(6)	Two pointers to elements of the same array object are subtracted, the result does not fit in the space provided and there is an arithmetic overflow.	Undefined		2.48

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
103	5.7(6)	Pointers that do not behave like pointers to elements of the same array are subtracted.	Undefined	Rule 5–0–17	2.49
104	5.7(6)	The signed integral type given as a result of the subtraction of two pointers to elements of the same array object.	Implementation		3.33
105	5.8(1)	An expression is shifted by a negative number or by an amount greater than or equal to the width in bits of the expression being shifted.	Undefined	Rule 5–8–1	2.50
106	5.8(3)	The value given as a result of >> shift operator where the *shift-expression* has a signed type and is negative	Implementation	Rule 5–0–21	3.34
107	5.9(2)	Pointers are compared using a relational operator that do not point to members of the same object, elements of the same array or to the same functions, etc…	Unspecified		1.13
108	5.17(8)	An object is assigned to an overlapping object.	Undefined	Rule 0–2–1	2.51
109	6.6.3(2)	The effect of flowing off the end of a function that is expected to return a value	Undefined	Rule 8–4–3	2.52
110	6.7(4)	Control re-enters a declaration recursively while an object is being initialized.	Undefined		2.53
111	6.8(3)	During parsing, a name in a template parameter is bound differently than it would be bound during a trial parse.	Behaviour that requires no diagnostics		5.07
112	7.1.5.1(4)	An attempt is made to modify a const object, other than any class member declared mutable.	Undefined	Rule 5–2–5	2.54
113	7.1.5.1(7)	An attempt is made to refer an object defined with volatile-qualified type through the use of an lvalue with non-volatile-qualified type.	Undefined	Rule 5–2–5	2.55
114	7.1.5.2(1)	Whether bit-fields and objects of char type are represented as signed or unsigned quantities.	Implementation	Rule 5–0–11 Rule 5–0–12 Rule 9–6–2	3.35
115	7.2(4)	The type of an uninitialized first enumerator.	Unspecified		1.15
116	7.2(4)	The value of an uninitialized enumerator is not representable in the type of the preceding enumerator.	Unspecified		1.16
117	7.2(5)	The integral type used as the *underlying type* for an enumeration.	Implementation		3.36
118	7.2(9)	A value is not in the range of the enumeration type to which it is explicitly converted.	Unspecified	Rule 7–2–1	1.17
119	7.3.2(4)	A namespace-name defined at global scope is also declared as the name of another entity in any global scope of the program.	Behaviour that requires no diagnostics		5.08
120	7.4(1)	The meaning of an *asm* declaration.	Implementation	Rule 7–4–1	3.37

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
121	7.5(1)	Implementation specific properties associated with an entity with language linkage	Implementation	Rule 1–0–2	3.38
122	7.5(2)	The meaning of the *string-literal* in a *linkage-specification*	Implementation	Rule 1–0–2	3.39
123	7.5(2)	The spelling of the language's name when the *string-literal* in a *linkage-specification* names a programming language	Implementation	Rule 1–0–2	3.40
124	7.5(2)	The semantics of a language linkage other than C++ or C.	Implementation	Rule 1–0–2	3.41
125	7.5(9)	Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages.	Implementation	Rule 1–0–2	3.42
126	8.3.2(3)	Whether a reference requires storage.	Unspecified		1.18
127	8.3.2(4)	Dereferencing a null pointer.	Undefined		2.56
128	8.3.6(9)	The order of evaluation of function arguments.	Unspecified		1.19
129	8.5(9)	The value of an object if no initializer is specified.	Indeterminate	Rule 8–5–1	4.04
130	8.5.3(8)	How the reference is bound when a reference to type "cv1 T1" is initialized by an expression "cv2 T2".	Implementation		3.43
131	9.2(12)	The order of allocation of non-static data members separated by an *access-specifier*	Unspecified		1.20
132	9.3.1(1)	A member function of a class X is called for an object that is not of type X or a type derived from X.	Undefined		2.57
133	9.6(1)	The allocation of bit-fields within a class.	Implementation	Rule 9–6–1	3.44
134	9.6(3)	Whether a plain (neither explicitly signed nor unsigned) *char*, *short*, *int* or *long* bit-field is signed or unsigned.	Implementation	Rule 9–6–2	3.46
135	9.6(4)	Alignment of bit-fields	Implementation		3.45
136	10(3)	The order in which the base class subobjects are allocated in the most derived object	Unspecified		1.21
137	10.3(8)	A virtual function declared in a class is both defined and declared pure in that class.	Behaviour that requires no diagnostics		5.09
138	10.4(6)	A virtual call is made from a constructor (or destructor) of an abstract class to a pure virtual function directly or indirectly for the object being created (or destroyed).	Undefined	Rule 12–1–1	2.58
139	11.1(2)	The order of allocation of data members with separate access-specifier labels	Unspecified		1.22

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
140	12.1(15)	The value of an object obtained, if during the construction of a const object, the object is accessed through an lvalue not obtained from the constructor's *this* pointer.	Unspecified		1.23
141	12.2(5)	The order of creation of temporary objects.	Unspecified		1.24
142	12.4(12)	A destructor is invoked for an object that is not of the destructor's class or not of a class derived from the destructor's class.	Undefined		2.59
143	12.4(14)	A destructor is invoked for an object whose lifetime has ended	Undefined		2.60
144	12.6.2(4)	The value of a member of a class if it is not otherwise initialized by the constructor.	Indeterminate		4.05
145	12.6.2(8)	A member function (including virtual member functions) is called for an object under construction, or an object under construction is used as the operand of the *typeid* operator or of a *dynamic_cast* performed in a ctor-initializer (or a function called directly or indirectly from a ctor-initializer) before all of the mem-initializers for base classes have been completed.	Undefined		2.61
146	12.7(1)	Referring to any nonstatic member or base class of an object of non-POD class type, before the constructor begins execution and after the destructor finishes execution.	Undefined		2.62
147	12.7(2)	Converting a pointer to an object of class X to a direct or indirect base class of X, where the construction of the object has not started or the destruction of the object has completed.	Undefined		2.63
148	12.7(2)	Forming a pointer to (or access the value of) a direct nonstatic member of an object, where the construction of the object has not started or the destruction of the object has completed.	Undefined		2.64
149	12.7(3)	The result of making a virtual call using an explicit class member access and the object expression refers to the object under construction or destruction but its type is neither the constructor or destructor's own class or one of its bases.	Undefined		2.65
150	12.7(4)	The operand of *typeid* refers to an object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases.	Undefined		2.66

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
151	12.7(5)	If the operand of the *dynamic_cast* refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor is not a pointer to or object of the constructor or destructor's own class or one of its bases.	Undefined		2.67
152	12.8(13)	Whether sub-objects representing virtual base classes are assigned more than once by the implicitly-defined copy assignment operator.	Unspecified		1.25
153	12.8(4)	Any use of a user defined copy constructor that matches the implicitly declared copy constructor	Behaviour that requires no diagnostics		5.10
154	14(4)	The linkage of a template, a template explicit specialization or a class template partial specialization, if it is something other than C or C++.	Implementation		3.47
155	14(8)	A template that is exported more than once in a program.	Behaviour that requires no diagnostics		5.11
156	14(8)	A non-exported template which is neither defined in every translation unit in which it is implicitly instantiated nor explicitly instantiated in some translation unit	Behaviour that requires no diagnostics		5.12
157	14.3.3(2)	A specialization is not visible at the point of instantiation, and it would have been selected had it been visible.	Behaviour that requires no diagnostics		5.13
158	14.5.4(1)	A partial specialization of a template is not declared before its first use that would cause implicit instantiation in any translation unit.	Behaviour that requires no diagnostics	Rule 14–7–3	5.14
159	14.5.5.1(7)	A program contains declarations of function templates that are functionally equivalent but not equivalent.	Behaviour that requires no diagnostics		5.15
160	14.6(7)	No valid specialization can be generated for a template definition, but the template is not instantiated.	Behaviour that requires no diagnostics		5.16
161	14.6.4.1(7)	Two different points of instantiation give a template specialization different meanings according to the one definition rule.	Behaviour that requires no diagnostics	Rule 14–7–3	5.17
162	14.6.4.2(1)	If a function call that depends on a template parameter would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced with those namespaces in all translation units	Undefined		2.68
163	14.7.1(14)	The instantiation of a template produces recursion beyond some defined limit	Undefined		2.69
164	14.7.1(14)	The limit on the total depth of recursive instantiation of templates	Implementation		3.48

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
165	14.7.1(5)	Whether the instantiation occurs when the overload resolution process can determine the correct function to call without instantiating a class template definition.	Unspecified		1.26
166	14.7.1(9)	Whether an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated.	Unspecified		1.27
167	14.7.3(6)	An explicit specialization of a template is not declared before its first use in any translation unit that causes implicit instantiation	Behaviour that requires no diagnostics	Rule 14–7–3	5.18
168	15.1(4)	The way memory is allocated for the temporary copy of an exception being thrown	Unspecified		1.28
169	15.1(4)	Deallocation of memory for a temporary object when the last handler exits by any means other than a throw and the temporary object is then destroyed.	Unspecified		1.29
170	15.3(10)	Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object.	Undefined	Rule 15–3–3	2.70
171	15.3(16)	Flowing off the end of a *function-try-block* in a value returning function.	Undefined		2.71
172	15.3(9)	Whether or not the stack is unwound before the call to *terminate()*, in the case where no matching handler is found in a program.	Implementation	Rule 15–1–3 Rule 15–3–1 Rule 15–3–2 Rule 15–3–4 Rule 15–5–1	3.49
173	15.4(2)	Sets of *type-ids* in exception-specifications in two translation units differ.	Behaviour that requires no diagnostics	Rule 15–4–1	5.19
174	15.5.2(2)	The object of type *std::bad_exception* that is used to replace an exception thrown or re-thrown by the *unexpected()* function that the *exception-specification* does not allow.	Implementation		3.50
175	16.1(4)	The token defined is generated during the expansion of a #if or #elif pre-processing directive.	Undefined		2.72
176	16.1(4)	The *#defined* pre-processing directive does not match one of the two specified forms.	Undefined	Rule 16–1–1	2.73
177	16.1(4)	Whether the value of an interpreted character literal matches the value obtained when an identical character literal occurs in an expression.	Implementation		3.51
178	16.1(4)	Whether a single-character character literal may have a negative value.	Implementation		3.52

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
179	16.2(2)	The sequence of places searched for the header file specified between the < and > delimiters due to a *#include <h-char-sequence>* new-line pre-processing directive.	Implementation		3.53
180	16.2(2)	During execution of a *#include* pre-processor directive, how the places are searched and how the header file is identified.	Implementation		3.54
181	16.2(3)	The sequence of places searched for the header file specified in quotes in a *#include "q-char-sequence" new-line* pre-processing directive.	Implementation		3.55
182	16.2(4)	The *#include* pre-processing directive that results after expansion does not match one of the header name forms.	Undefined	Rule 16–2–6	2.74
183	16.2(4)	The method by which a sequence of pre-processing tokens between < and > or a pair of " characters is combined into a single header name pre-processing token.	Implementation		3.56
184	16.2(5)	The mapping between the delimited sequence and the external source file name.	Implementation		3.57
185	16.2(6)	The nesting limit to which an *#include* pre-processing directive may appear due to the *#include* directive of another file.	Implementation		3.58
186	16.3(10)	A function-like macro argument consists of no pre-processing tokens.	Undefined	Rule 16–0–4	2.75
187	16.3(10)	There are sequences of pre-processing tokens within the list of function-like macro arguments that would otherwise act as pre-processing directive lines.	Undefined	Rule 16–0–5	2.76
188	16.3.2(2)	The order of evaluation of # and ## operators.	Unspecified	Rule 16–3–1 Rule 16–3–2	1.30
189	16.3.2(2)	The result of the pre-processing operator # is not a valid character string literal.	Undefined	Rule 16–3–1 Rule 16–3–2	2.77
190	16.3.3(3)	The order of evaluation of ## operators.	Unspecified	Rule 16–3–1 Rule 16–3–2	1.31
191	16.3.3(3)	The result of the pre-processing concatenation operator ## is not a valid pre-processing token.	Undefined	Rule 16–3–1 Rule 16–3–2	2.78
192	16.4(3)	The #line pre-processing directive specifies zero or a number greater than 32767.	Undefined		2.79
193	16.4(5)	The #line pre-processing directive that results after expansion does not match one of the two well-defined forms.	Undefined		2.80
194	16.6(1)	The behaviour of the implementation due to the *#pragma* pre-processing directive.	Implementation	Rule 16–6–1	3.59
195	16.8(1)	The date/time supplied, as a result of the *__DATE__* macro, if the date of translation is not available.	Implementation		3.60

MISRA Id	ISO Reference	Description	Category	MISRA Guidance	Source Id
196	16.8(1)	The date/time supplied, as a result of the *__TIME__* macro, if the time of translation is not available.	Implementation		3.61
197	16.8(1)	Whether *__STDC__* is predefined and its value.	Implementation		3.62
198	16.8(3)	One of the following identifiers is the subject of a *#define* or a *#undef* pre-processing directive. *__LINE__*, *__FILE__*, *__DATE__*, *__TIME__*, *__STDC__*, *__cplusplus*, or the identifier *defined*.	Undefined	Rule 17–0–1	2.81

Appendix C

Appendix C: Glossary

ADL

ADL is an abbreviation for *argument-dependent lookup*.

Aggregate

An array is an *aggregate*. A class is an *aggregate* if all the following apply:

- It has no user declared constructors;
- It has no private or protected non-static data members;
- It has no base classes;
- It has no virtual functions.

Associated namespace

An *associated namespace* is an additional namespace that is searched during *argument-dependent lookup*. The set of *associated namespaces* is described in ISO/IEC 14882:2003 [1] §3.4.2(2).

Callback

A *callback* is a function that is called indirectly via a function pointer or a handle.

Class-data

The *class-data* for a class is all non-static member data and any *resources* acquired in the constructor or released in the destructor.

Code

Code consists of everything within a translation unit that is not excluded by conditional compilation. Note that *code* also includes any declarations and definitions that are introduced by the compiler (e.g. default constructors, etc).

Compatible types

Compatible types are types that, for the purpose of declaration matching, are treated as the same. Two identical types are compatible but two *compatible types* need not be identical. For example, *short int* and *short* are compatible.

Dataflow anomaly

The state of a variable at a point in a program can be described using the following terms:

- Undefined (U): The value of the variable is indeterminate;
- Referenced (R): The variable is used in some way (e.g. in an expression);
- Defined (D): The variable is explicitly initialised or assigned a value.

Given the above, the following *dataflow anomalies* can be defined:

- **UR dataflow anomaly**: Variable not assigned a value before the specified use;
- **DU dataflow anomaly**: Variable is assigned a value that is never subsequently used;
- **DD dataflow anomaly**: Variable is assigned a value twice with no intermediate use.

DD dataflow anomaly

See *dataflow anomaly*.

Dead code

Dead code (also known as *redundant code*) consists of evaluated expressions whose removal would not affect the output of a program.

Declaration

For the purposes of this standard, in headline rule text a *declaration* is the first introduction of a name into a translation unit. All subsequent "declarations" (as per ISO/IEC 14882:2003 [1] §3.2(1)) are *re-declarations*.

A *definition* is also always either a *declaration* or *re-declaration*, but with the characteristics described in ISO/IEC 14882:2003 [1] §3.2(2).

Definition

See *Declaration*.

DU dataflow anomaly

See *dataflow anomaly*.

Function set

A *function set* is:

- A function;
- A function overload set.

Generic function

A function template or operator template that can be called without explicit template arguments and whose parameters are of built-in type or which are *generic parameters*. For example:

```
template <typename T> void f(T const & t);
```

Generic parameter

A *template type parameter* T is a *generic parameter* if, in the function declaration, it has the (possibly *cv-qualified*) form `T &`$_{[opt]}$.

For example:

```
T const & t
T
T volatile
```

Handle

A *handle* is a variable that refers to a *resource*.

Header file

A *header file* is any file that is the subject of a *#include* directive. Note that the filename extension is not significant.

Include guard

An *include guard* is a construct used to avoid the problems associated with multiple inclusion when dealing with the *#include* directive.

Infeasible path

Infeasible paths occur where there is a syntactic path to a code fragment, but the semantics ensure that the control flow path will not be executed. Example:

```
if ( u32 < 0 )
{
   // An unsigned value will never be negative,
   // so code in this block will never be executed.
}
```

LRM

LRM is an abbreviation for Language Reference Manual.

NDR

NDR is an abbreviation for No Diagnostic Required.

ODR

ODR is an abbreviation for the One Definition Rule.

POD

POD is an abbreviation for Plain Old Data.

A *struct* or *union* is POD if all the following apply:

- It is an aggregate;
- It has no non-static members with type non-POD (or array of non-POD);
- It has no non-static members with type reference;
- It has no user declared copy assignment operator;
- It has no user declared destructor.

A class is a POD if it is either a *struct* or a *union* that is a POD.

Project

A *project* consists of the *code* from the set of translation units used to build the application.

Primary template

The *primary template* is the first declaration of a template.

Redundant code

See *Dead code*.

Re-declaration

See *Declaration*.

Resource

A *resource* is an entity whose lifetime is controlled explicitly by the developer. The developer is therefore responsible for acquiring and relinquishing the *resource*.

Unreachable code

Unreachable code is code to which there is no syntactic (control flow) path, e.g. a function which is never called, either directly or indirectly.

UR dataflow anomaly

See *dataflow anomaly*.

Use / used / using

An object is *used* if it is:

- The subject of a cast;
- Explicitly initialized at declaration time;
- An operand in an expression;
- Referenced.

Unique

Within the rules, *unique* means that the associated text/rule applies across the whole *project*.

Unused

A type or object is *unused* if it is not *used*.

ISBN 978-1-906400-03-3 paperback
ISBN 978-1-906400-04-0 PDF