SEC BOOKS

ESCR C++ Ver. 2.0

Revised Edition

ESCR C++
Embedded System development Coding Reference guide

[C++ Language Edition]

Written and edited by
Software Riliability Enhancement Center,
Technology Headquarters,
Information-technology Promotion Agency,Japan

IPA

This document is the English edition of ESCR (Embedded System development Coding Reference) [C++ language edition] Version 2.0 published by IPA/SEC* in Japan. It is the revised English edition of ESCR [C++ language edition] Version 1.0 made available in March 2013 in pdf format. Aimed at improving the quality of the source code written in C++ language, ESCR collects the important points to be noted as part of the know-how for coding and organizes them as practices and rules.

The purpose of this document is to be used as a reference guide for establishing coding conventions in organizations and groups developing embedded software using C++ language, and for promoting the standardization of coding styles and uniformity of source code quality.

October 2016

Software Reliability Enhancement Center, Technology Headquarters,

Information-technology Promotion Agency, Japan

* Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, Japan

Preface

On publication of ESCR [C++ language edition] Ver. 2.0

This document is the English edition of ESCR (Embedded System development Coding Reference) [C++ language edition] Version 2.0 published by IPA/SEC* in Japan. Aimed at improving the quality of the source code written in C++ language for software development, ESCR C++ has been organized as a compilation of important points and tips to keep in mind throughout the coding process.

ESCR C++ Ver. 2.0 is the updated version of ESCR C++ Ver. 1.0 that was released in November 2010. Since C++ is a language derived and extended from C language, ESCR C++ Ver. 1.0 follows the same structure used for ESCR C, and is based on the structure of the latest version of ESCR C, which is Ver. 1.1 released in 2007.

ESCR C++ Ver. 1.0 complies with ISO/IEC14882:2003 (C++03), which was the most widely used C++ language standard at the time Ver. 1.0 was published. Since then, C++ language standard has evolved and a number of updated versions have been introduced and are being used by increasing number of C++ language programs. Likewise, ESCR C that is used as the basis of ESCR C++ has also been updated in conjunction with the introduction of updated versions of C language standard and MISRA C. To respond to these developments, we have come to release ESCR C++ Ver. 2.0 with an intention of meeting the following two objectives:

- Make the rules compliant with the updated versions of C++ language standard, C++11 and C++14, and add new rules to make it easy for the programmers to use the new features supported by these updated versions;
- Make ESCR C++ Ver. 2.0 consistent with ESCR C Ver. 2.0

To maintain the continuity from the previous version, ESCR C++ Ver. 2.0 follows the same structure as Ver. 1.0. The practices and rules carried over from the previous version are also numbered the same as in Ver. 1.0. To support the language specifications that have been extended in C++11 and C++14, various descriptions have been added to Ver. 2.0, including the introduction of new rules, supplementary explanatory texts, and additional compliant and non-compliant coding examples.

Our objective of publishing ESCR C++ Ver. 2.0 remains the same as the previous version. This document is intended to serve as a source of reference for organizations and groups in need of establishing coding rules to be strictly followed by their software developers for the purpose of standardizing and maintaining uniform level of quality of their source code written in C++ language.

Lastly but not least, we would like to thank the members of Coding Practices Guide Revision Working Group for their cooperation in reviewing ESCR C++ Ver. 1.0 closely based on the latest versions of C++ language standard, C++11/C++14 and the contents of ESCR C Ver. 2.0, and Honorary Professor Emeritus Ikuo Nakata of Tsukuba University for his guidance in identifying the information that need to be added, revised or deleted in or from this document to ensure that it is up to date at the time of its publication.

We sincerely hope that the effective use of ESCR C++ Ver. 2.0 will lead to improved productivity of embedded software and contribute to the attainment of high-quality software development.

October 2016

Yukihiro Mihara, Keisuke Toyama
Software Reliability Enhancement Center, Technology Headquarters,
Information-technology Promotion Agency, Japan

Fusako Mitsuhashi
Coding Practices Guide Revision Working Group

Table of Contents

Part 1

How to Read the Coding Practices Guide

1 Overview

1.1 What are Coding Practices?

Creating source code (code implementation) is an inevitable task for developing embedded software. Success or failure of this task greatly affects the quality of the resulting software. C and C++, the two programming languages that are very commonly used for embedded software development, are said to give the programmers a relatively extensive writing flexibility. The quality of programs written in C or C++ thus tends to reflect quite clearly the difference in coding skill level between seasoned and less-experienced programmers. It is undesirable to have source code varying largely in quality, depending on the programmers' individual coding skills and experience. To prevent this risk from leading into serious quality issues, forward-thinking companies are working proactively toward standardization of their source codes by establishing coding standards or conventions to be followed organization-wide or group-wide.

Issues Regarding Coding Conventions

Coding convention is generally regarded as the organized set of "styles of (or rules in) writing code that need to be followed to maintain quality." However, it is becoming a common understanding that various issues exist in the current usage of coding conventions, including those mentioned below.

1) The necessity of rules is not understood. The appropriate methods to deal with rule violations are also not widely shared.
2) There are too many rules to learn. Yet, the existing rules are not comprehensive enough to cover the entire scope of coding.
3) Since highly reliable tools that can thoroughly and accurately check whether the written code is complying with the relevant rules or not are unavailable, the engineers have to review the code manually through visual check, which is a heavy burden for them.

Due to such circumstances, there are, in fact, some coding conventions established at the organization or department level that have lost their significance and are no longer strictly observed.

Nevertheless, organizations that have coding conventions, no matter what kind of format they prepare them in, are at least better than those without any. There are still quite a few that cannot reach a consensus of the coding convention to be followed internally, and are relying largely on the programmers' individual judgments to decide how the source code should be written.

What are Coding Practices?

This guide aims at solving on-site issues related to coding conventions, by providing a collection of practical coding techniques considered important from the standpoint of software quality that conform to the basic way of thinking (concept) to be followed in various coding situations. They are referred to as "coding practices" in this guide, and are presented with detailed description and specific examples of related coding conventions (or rules) for reference.

This guide is intended to enable the users to solve the above-mentioned issues in coding conventions by "establishing a concrete and effective coding convention for their own organization", using the set of relevant information provided herein as their reference.

1.2 Purpose and Position of Coding Practices and Target Users

Purpose and Position of Coding Practices

ESCR [C++ language edition] Ver. 2.0 is a guide on coding practices intended to help enable those who create and/or operate the coding conventions to establish them in their companies or projects. This book is characteristic for regarding coding conventions as "ways of writing code that should be followed by all the programmers in a given project to maintain quality" and organizing the basic rule concepts as practices. These practices are broken down into outline and details, based on the quality concept that complies with "ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality model" They are respectively explained with corresponding C++ programming rules and the rationale for using them. Through these practices and rules, ESCR [C++ language edition] Ver. 2.0 aims at enabling the users to easily establish their own "coding convention" that meets their practical needs and also clearly explain why the included practices and rules are significant and necessary.

Target Users

This guide has been written on the assumption that it will be read by the following types of users:

Creators of the Coding Conventions

This book can be used as a reference guide to create a new coding convention or to review and re-organize existing coding conventions.

Programmers and Program Reviewers

Highly reliable and maintainable code can be produced with reasonable effort by learning and understanding the practices and rules provided in this guide.

Benefits Gained

The benefits that the users can expect to gain directly from using this guide are as mentioned above. Moreover, as a result of these benefits, the users may also be able to expect the following positive effects:

- Can remove the bottleneck in maintaining software quality caused by inconsistent performance of implementation engineers;
- Can eliminate obvious errors in the source code at an early stage, such as, during the coding phase or in subsequent reviews.

1.3 Characteristics of the Coding Practices

The coding practices in this guide have the following characteristics.

Systematically Organized Practices and Rules

This guide considers that code quality can also be classified like software quality, according to quality characteristics, such as, “reliability”, “maintainability” and “portability”, and organizes the coding practices and rules systematically based on “ISO/IEC9126-1:2001 Software engineering - Product quality”. The coding practices described in this guide are customs and ideas on implementation that have been developed to maintain source code quality, and they reflect the basic concepts of individual coding rules. The coding rules included in this guide have been selected based on the needs of the current conditions (actual situation with the language specifications and processing systems) after closely examining the various coding conventions existing in the world, and are presented in the form of established information that supports the corresponding practices. Classification of practices and rules according to quality characteristics makes it easy for the users of this guide to understand their respective purpose of use in terms of which aspect of quality each of them is primarily focused on maintaining.

The coding conventions referenced in this guide range from local conventions used in companies to which the writers and reviewers of this guide belong, to sets of coding rules established and used widely in different industries, including “MISRA C++”. For details, refer to “Appendix: Bibliography” at the end of this book.

Ready-to-use Reference Rules

This guide presents specific rules for C++ language as reference information for creating coding conventions. These rules can be used directly as coding conventions. By referring to “3. How to Use this Guide”, the users of this guide can easily create their own coding conventions for C++ language by choosing the rules that meet their respective needs and adding any other rules that they feel are also necessary to cover the areas that are not sufficiently addressed.

Presenting the Necessity of Each Rule

The coding rules covered in this guide are respectively described with explanation of corresponding practices and examples of how the code should be written (to be compliant with the given rule) or should not be written (because that would be non-compliant), to enable the users of this guide to understand clearly why each of them is necessary. Rules considered to be already well-known among experienced programmers are so indicated in the “Preference guide” to help the users of this guide determine whether they need to include these rules in their conventions or not.

Correspondence with Other Coding Conventions

In preparing this guide, a study was conducted to find the correspondence relationship between each coding rule and coding conventions used widely in various industries. In particular, the rules in “MISRA-C” and “MISRA-C++” that are referenced in this guide have all been extracted and organized in the form of a table titled “List of practices and rules”, which is available in the Appendix section at the end of this book.

1.4 Notes on Using this Guide

Keep the following points in mind when using this guide.

Scope of the Rules

This guide conforms to C++ language specifications, but the rules related to any of the following are considered out of scope of C++ language reference rules:

- Library functions;
- Metrics (numbers of lines in functions/complexity of functions, etc);
- Errors in writing possibly classified as coding errors;
- Rules for creating templates.

In this guide, the descriptions under "Part 3 Typical Coding Errors in Embedded Software" in ESCR [C language edition] have been entirely omitted. Those interested in knowing what was written in there should refer to ESCR [C language edition]. Since many examples of coding errors described in there can also be a good reference for C++ programmers, especially the beginners in C++ programming are strongly advised to read them through.

Cited or Referenced Standards in this Guide

In this guide the following standards have been cited or referenced.

C90

This is the C language standard defined in "ISO/IEC 9899:1990 Programming Language C". It is often called C90, where "90" stands for the year ISO/IEC 9899:1990 was published. The C language standard has been revised and is now C99, making C90 an older version.

C99

This is the C language standard defined in "ISO/IEC 9899:1999 Programming Language C". It is the current standard widely used. Since ISO/IEC 9899:1999 was published in 1999, it is often called "C99".

C11

This is the most recent C language standard defined in "ISO/IEC 9899:2011 Programming Language C" and thereby is the current C language standard. Since ISO/IEC 9899:2011 was published in 2011, it is often called "C11".

C++03

This is the C++ language standard defined in "ISO/IEC 14882:2003 Programming language C++". It is often called "C++03".

C++11

This is the C++ language standard defined in "ISO/IEC 14882:2011 Programming language C++". It is often called "C++11".

C++14

This is the C++ language standard defined in "ISO/IEC 14882:2014 Programming language C++". It is often called "C++14". The changes from previous version are all minor.

MISRA C, MISRA C++

MISRA C refers collectively to the coding guidelines for C language defined by The Motor Industry Software Reliability Association (MISRA) in UK, which include MISRA C:1998, MISRA C:2004 and MISRA C:2012. MISRA C++ refers to the coding guidelines for C++ language defined by MISRA in UK, which include MISRA C++:2008.

MISRA C: 1998

The convention in Citations and References [9].

MISRA C: 2004

The convention in Citations and References [10]. This is the revised version of MISRA C:1998.

MISRA C2012

The convention in Citations and References [11]. This is the revised version of MISRA C:2004.

MISRA C++: 2008

The convention in Citations and References [12].

Difference from Ver. 1.0

This document (Ver. 2.0) has been updated from the previous version (Ver. 1.1) primarily by adding rules considered to be necessary when using the new features defined mainly in C++11, and by modifying some rules and descriptions considered necessary to do so in order to maintain consistency with the revisions made in other published versions of ESCR. There are some descriptions that deal with the same matters explained in Ver. 1.0, but have been rewritten in this document with more clarity or in a way that would be easier to understand.
The rule number of each deleted rule is treated as missing number and kept vacant. The rules that deal with the same topic in Ver. 1.0 are numbered the same in this document so that the users of old ESCR C++ edition (Ver 1.0) can find it easy to track the rules.

The rules that have been newly added to this document are R3.6.3, R3.7.7, R3.10, R3.10.1, R3.11, R3.11.1, R3.11.2, M2.3, M2.3.1, M4.7.7, P1.6, and P1.6.1. The rules in Ver. 1.0 that have been deleted in the document are M3.1.3 and M4.7.4.

Column: Points to be careful of when using the new features in C++

C++ language specifications defined in C++03 have been expanded in many ways through the new features introduced in C++11 and C++14. Many of these new features are intended to improve the productivity and reliability of the program, and can enhance the quality of software developed in C++ language when used effectively. In this document, you can find coding rules that should be observed when using some of these new features that contribute to the extended usage of C++ language. But there are still many newly introduced features that require deep knowledge to utilize them properly, including those that do not have clearly established coding rules yet. Therefore, the users of these new features must keep in mind to always use them with utmost care.

Provided below are two examples of new features that need to be used very carefully.

(1) Type inference of a variable

In the conventional C++ language standard, the keyword auto referred to an automatic variable, but was almost never used. In C++11, the specifications have been modified to make the compiler infer the type of a variable from the type of the initial value when auto is written in lieu of the variable type in the variable declaration where the initial value is specified. For example, if

```
auto x = f();
```

the type of variable x will be the same as the return value type of function f. Type inference by auto is useful to prevent the code from becoming redundant in cases where complex types are used. But auto also has a risk of making the compiler infer an unexpected type as the variable type when the code is modified. Therefore, there is a need to use auto carefully to prevent incorrect inferences from being made.

(2) Function declaration form

In the conventional form of function declaration, the return value type of a function was written before the function name. The language features expanded by C++11 also support the declaration form of writing the return value after the parameter declaration. According to C++11, the conventional function declaration,

```
int f(int);
```

can also be declared in the following form:

```
auto f(int) -> int;
```

The newly supported form of function declaration is useful for being able to infer the return value type from the parameter type, such as, in template definition. But to prevent confusion among the members of the software development project, it is highly recommended to set local rules to be followed by the project members on how to use this additionally supported function declaration form and when it is applicable.

2. Understanding Source Code Quality

2.1 Quality Characteristics

For many, speaking of software quality would remind them of "bugs." However, in the field of software engineering, the quality of software as a product is grasped in a broader perspective. This concept of software product quality is defined in detail and organized systematically in ISO/IEC 25010:2011.

ISO/IEC 25010:2011 and Source Code Quality

ISO/IEC 25010:2011 defines the quality of software product by breaking it down into eight characteristics (quality characteristics): "reliability", "maintainability", "portability", "efficiency", "security", "functionality", "usability" and "compatibility". Among them, "functionality", "usability" and "compatibility" are considered to be the three quality characteristics that should be addressed at an early stage, preferably before moving on to the design phases in the upstream process. Whereas, "reliability", "maintainability", "portability", and "efficiency" are considered to be the quality characteristics that have close relevance with the development of high-quality source code and should therefore be examined in depth during the coding phase. "Security", which has been defined as the quality subcharacteristic of "functionality" in the previous standard, ISO/IEC 9126-1, is considered basically as a quality characteristic that is relevant in the design phase, but coding such as for avoiding stack overflow can also affect security. For more information on coding practices related to security, please refer to CERT C Secure Coding Standard [2].

Based on the above broad categorization, this guide has adopted the latter four quality characteristics - "reliability", "maintainability", "portability", and "efficiency" - as the main focus, and gathered the coding practices that are primarily concerned with any of these four. Table 1 shows the relationship between the "quality characteristics" defined in ISO/IEC 25010 and the "code quality" proposed in this guide, along with the "quality subcharacteristics".

Table 1 Quality Characteristics of Software and Code Quality

Quality characteristics (ISO/IEC 25010)		Quality subcharacteristics (ISO/IEC 25010)		Code quality
Reliability	Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time	Maturity	Degree to which a system meets needs for reliability under normal operation	Low occurrence of bugs through continued use
		Availability	Degree to which a system, product or component is operational and accessible when required for use	
		Fault Tolerance	Degree to which a system, product or component operates as intended despite the presence of hardware or software faults	Tolerance for bugs and interface violations, etc
		Recoverability	Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired	
Maintainability	Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers	Modularity	Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components	Degree to which the components are composed such that a change to one component of the code has minimal impact on other components
		Reusability	Degree to which an asset can be used in more than one system, or in building other assets	Degree to which a code can be used in other programs
		Analysability	Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified	Easiness of understanding the code
		Modifiability	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality	Easiness of modifying the code, and lowness of impact from modifications
		Testability	Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met	Easiness of testing and debugging the modified code

Quality characteristics (ISO/IEC 25010)		Quality subcharacteristics (ISO/IEC 25010)		Code quality
Portability	Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another	Adaptability	Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments	Easiness of adapting to different environments *Including conformance to standards
		Installability	Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment	
		Replaceability	Degree to which a product can be replaced by another specified software product for the same purpose in the same environment	
Performance Efficiency	Performance relative to the amount of resources used under stated conditions	Time Behaviour	Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements	Efficiency with regard to processing time
		Resource Utilization	Degree to which the amounts and types of resources used by a product or system when performing its functions meet requirements	Efficiency with regard to resources
		Capacity	Degree to which the maximum limits of a product or system parameter meet requirements	
Security	Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization	Confidentiality	Degree to which a product or system ensures that data are accessible only to those authorized to have access	Degree of certainty that data are accessible only to those authorized to have access
		Integrity	Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data	Degree of prevention of unauthorized access to, or modification of, computer programs or data
		Nonrepudiation	Degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later	
		Accountability	Degree to which the actions of an entity can be traced uniquely to the entity	
		Authenticity	Degree to which the identity of a subject or resource can be proved to be the one claimed	

2.2 Quality Characteristics, and Coding Practices and Rules

Overall Structure

In this guide, the basic matters to be followed when creating source code are organized as “practices”. For each “practice”, this guide introduces “rules” that are more specific reference information to keep in mind at the time of coding.

These “practices” and “rules” provided in this guide are classified and arranged in order, according to their association to any of the four quality characteristics described earlier in 2.1. The following section defines what “practice” and “rule” actually mean in this guide (see also Figure 1):

Practice

A “practice” is a custom or a set of ideas on implementation to maintain source code quality. Each practice reflects the basic concept of individual coding rule. These practices are broken down into outline and details.

Rule

A “rule” is a specific agreement that must be followed and constitutes a part of coding convention. This guide presents these rules as reference information. In this guide, a “rule” is also sometimes used as a collective term that represents a group of relevant rules.

Correspondence of Practices and Rules

Most practices and rules are related to multiple quality characteristics, but in this guide, they are respectively discussed in the section of the characteristic to which they are most strongly related. Associating each practice with a particular quality characteristic makes it possible and easy for the users of this guide to understand how each practice strongly affects which aspect of quality.

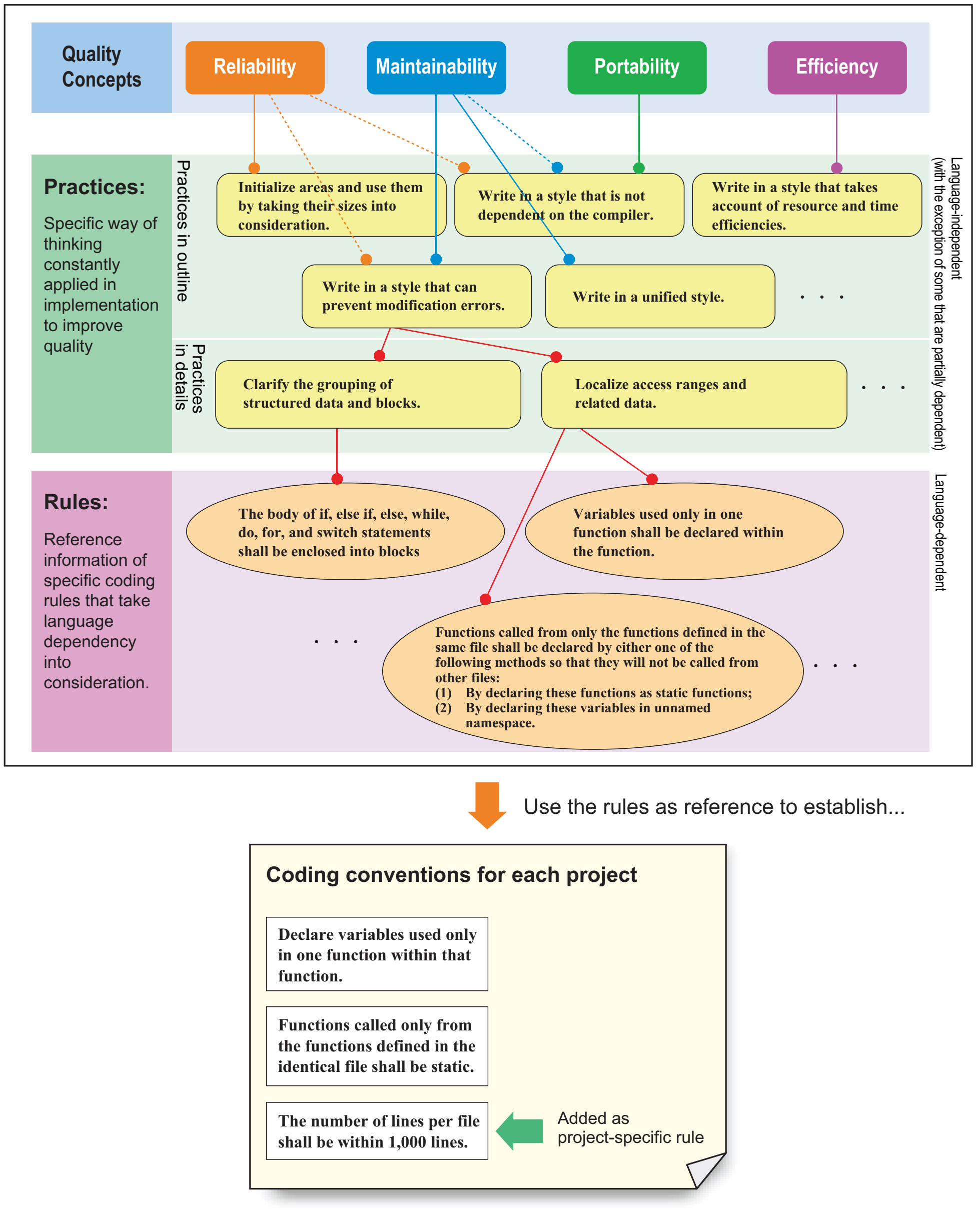

Quality Concepts
Reliability
Maintainability
Portability
Efficiency
Practices:
Specific way of thinking constantly applied in implementation to improve quality
Practices in outline
Initialize areas and use them by taking their sizes into consideration.
Write in a style that is not dependent on the compiler.
Write in a style that takes account of resource and time efficiencies.
Write in a style that can prevent modification errors.
Write in a unified style.
. . .
Practices in details
Clarify the grouping of structured data and blocks.
Localize access ranges and related data.
. . .
Language-independent (with the exception of some that are partially dependent)
Rules:
Reference information of specific coding rules that take language dependency into consideration.
The body of if, else if, else, while, do, for, and switch statements shall be enclosed into blocks
Variables used only in one function shall be declared within the function.
Functions called from only the functions defined in the same file shall be declared by either one of the following methods so that they will not be called from other files:
(1) By declaring these functions as static functions;
(2) By declaring these variables in unnamed namespace.
. . .
. . .
Language-dependent
Use the rules as reference to establish...
Coding conventions for each project
Declare variables used only in one function within that function.
Functions called only from the functions defined in the identical file shall be static.
The number of lines per file shall be within 1,000 lines.
Added as project-specific rule

Figure1. Relationship Between Quality Concepts, Practices, and Rules

3. How to Use this Guide

3.1 Scenarios for Using this Guide

Usage Scenarios

This guide intends to support the creation of coding conventions, and assumes that it will be used in the following three scenarios:

1) Creating a new coding convention;
2) Enhancing existing coding conventions;
3) Serving as a learning material for programmers' training and self-study.

Creating a New Coding Convention

Organizations or departments that have not been able to organize any coding conventions to be followed internally can use this guide for reference to establish their own coding convention that suits their respective needs.

Enhancing Existing Coding Conventions

Even in organizations and departments that have already established their coding conventions, it is effective to maintain them regularly. Using this guide as a reference will help them review the contents of their existing coding conventions more efficiently.

Serving as a Learning Material for Programmers' Training and Self-Study

There are many books published on C++ language. Unlike those existing ones, this guide focuses on implementation quality, and provides an organized set of information on how to create source code that can maintain and improve its quality. In this sense, this guide can also be an excellent material for the users to learn about source code quality from a more practical point of view.

3.2 Creating a New Coding Convention

This section presents the procedure for creating a new coding convention by using this guide. It is intended for projects that do not have any coding conventions of their own.

When to Create

Create the coding convention before proceeding to the program design stage. While a coding convention is a group of rules that are referred to during coding, some rules, such as, the naming convention applied to function names are associated with program design, and therefore need to be decided before starting the program design.

How to Create

Projects creating a new coding convention of their own are recommended to follow the procedure described below, step by step:

Step-1	Decide on the policy for creating a coding convention.
Step-2	Choose the rules based on the creation policy that has been decided.
Step-3	Define the project-dependent parts of the rules.
Step-4	Determine the procedure for setting exceptions to the rules if necessary.

After following these steps in order, add any other rules as needed.

Step-1 Decide on the policy for creating a coding convention

In creating a coding convention, the first thing to do is to decide on its policy. A creation policy defines how the code should be written for the project, based on, such as, the characteristics of the software developed in the project and the members of the project. For example, should the priority be placed on safety and write code that avoids using features that are not safe, even if they are convenient to use? Or, should the code be written in a way that makes careful use of such unsafe but convenient features? These are some of the questions that need to be addressed in the creation policy. When deciding on the creation policy, each project should consider which quality characteristics are particularly important for its software development, and examine what kind of coding practices it should adopt from the following perspectives:

- Coding that takes account of fail-safe;
- Coding that improves the program readability;
- Coding that makes debugging easy, etc.

Step-2 Choose the rules based on the creation policy that has been decided

The next step is to choose the suitable rules from the Practices Chart in Part 2, based on the creation policy decided in Step-1. If the project decides on the policy that prioritizes on portability, for example, efforts should be made to include many rules that address the portability issues in its coding convention.

In "Part 2 - Coding Practices for Embedded Software" of this guide, some rules are marked with either "○" or "●" as a guide to facilitate the selection process. (For more information, see the description under "Preference guide" in Part 2.) A rule marked with "○" indicates that it is regarded so important for the particular quality characteristic it addresses, that if this rule is not adopted as a part of the coding convention, that quality aspect may be seriously impaired. Whereas, "●" indicates that it is a rule that is already so well-known among those who are very knowledgeable about C++ language specifications that it may not necessarily be included in the coding convention. The simplest way of creating a coding convention would therefore be to choose only the rules indicated with "○", which would result in a set of widely applied rules.

Step-3 Define the project-dependent parts of the rules

In this guide, the rules are treated as one of the following three types:

1) Rules that can be used as a part of the coding convention without making any changes (In the "Rule specification" field, these rules are not marked.)
2) Rules that need to be chosen from several alternatives, according to the project characteristics (In the "Rule specification" field, these rules are marked as "Choose".)
3) Rules that need to be prescribed more specifically in a document (In the "Rule specification" field, these rules are marked either as "Define" or "Document".)

The rules treated either as type 2) or type 3) cannot be included in the coding convention as they are. For rules treated as type 2) to be adopted as a part of the newly created coding convention, they must be first chosen from the multiple alternatives presented in this guide. To adopt the rules treated as type 3) as a part of the coding convention, they must be more fine-tuned so that they can address the specific needs of each project. In doing so, the supplementary explanation provided to each practice described in this guide should serve as a useful reference on rule definition.

Step-4 Determine the procedure for setting exceptions to the rules

The quality characteristics that should be focused at the time of coding may differ, depending on the feature the project is intending to realize through implementation. (For example, "In this project, efficiency should be prioritized over maintainability…"). There may be cases when writing code that is fully compliant with a certain rule included in the coding convention causes difficulty in achieving the project-specific objective. To deal with such cases, it is necessary to have a procedure to allow partial exceptions to this rule.

The important points to be covered in this procedure are as follows:

- Describe what problems may occur by writing code that is compliant with the rule;
- Have experts review the problems and possible solutions;
- Record the review result.

Be sure not to allow exceptions too easily. The substance of the rule will be lost when there are too many exceptions.

The following is an example of the procedure for allowing exceptions.

[Example procedure]

(1) Prepare a form describing the reason for the exception.
(This form should, for example, contain the following items.)
- Rule number;
- Location of the code at issue (file name, line number);
- Problem(s) caused by complying with the rule;
- Impact of deviation from the rule.

(2) Have experts review the problems and possible solutions. → Enter the review result in the form.

(3) Gain approval from the head person (manager, project leader, etc) responsible of the coding process. → Record the approval in the form.

3.3 Enhancing Existing Coding Conventions

For projects where coding conventions already exist, this guide can be a useful reference to review and enhance the contents of their coding conventions.

Preventing Oversights and Omissions

By sorting the rules in existing coding conventions based on the concept of practices described in this guide, the project members will be able to identify and supplement the elements that have been overlooked or omitted, and see in a fresh light which tasks they have been placing importance on in their project.

Clarifying the Necessity of Rules

For those who have been feeling compelled to follow some rules without knowing why, this guide will serve as a useful tool to understand clearly why they are necessary by referring to the practices and compliant examples showing how they should be used.

3.4 Serving as a Learning Material for Programmers' Training and Self-Study

This guide is a good learning material for programmers who have studied C++ language but are still not used to or have little experience in practical coding.

Target Users

This guide is targeted at the following group of programmers:

- Programmers who have studied and acquired the basic skills in C++ language
- Programmers who have experience in other programming languages but are beginners in C++ language

What the Users Can Learn

By reading this guide, which is organized from the standpoint of quality characteristics like reliability, maintainability and portability, the users can learn:

- How to write code that can improve reliability;
- How to write code that can prevent bugs from being produced;
- How to write code that can facilitate debugging and testing;
- How to write code that is easy to read and why good readability is necessary

Part 2

Coding Practices for Embedded Software: Practices Chart

How to Read the Practices Chart

Organizational Structure of the Practices

Coding practices shown in Part 2 are classified according to four software quality characteristics (reliability, maintainability, portability, efficiency).

Practices in Outline

Practices closely related to each characteristic are largely divided into “practices in outline”. For example, the practices closely related to maintainability are largely divided into five practices in outline from "Maintainability M1: Keep in mind that others will read the program" to "Maintainability M5: Write in a style that makes testing easy".

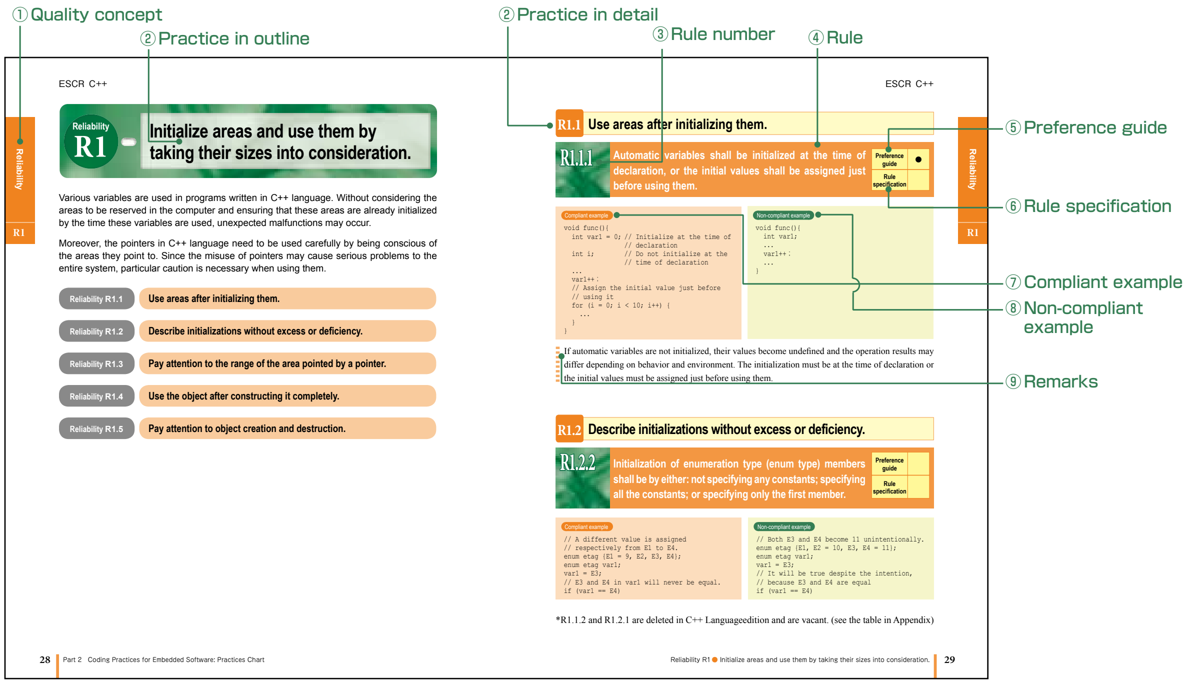
Practices in Detail

Each practice in outline is broken down into more specific subsets called “practices in detail”. For example, the practice in outline "Maintainability M3: Write programs simply" has five practices in detail, which are:

Maintainability M3.1	**Do structured programming.**
Maintainability M3.2	**Limit the number of side effects per statement to one.**
Maintainability M3.3	**Write expressions that differ in purpose separately.**
Maintainability M3.4	**Do not use complicated pointer operation.**
Maintainability M3.5	**Do not use complicated class structure.**

Layout of the Practices Chart

For each practice, reference information on rules to be noted during actual coding is provided in a chart form. The following diagram shows the layout of a sample chart, which is followed by the description of each field composing the chart:



ESCR C++

Reliability R1 — Initialize areas and use them by taking their sizes into consideration.

Various variables are used in programs written in C++ language. Without considering the areas to be reserved in the computer and ensuring that these areas are already initialized by the time these variables are used, unexpected malfunctions may occur.

Moreover, the pointers in C++ language need to be used carefully by being conscious of the areas they point to. Since the misuse of pointers may cause serious problems to the entire system, particular caution is necessary when using them.

Reliability R1.1	Use areas after initializing them.
Reliability R1.2	Describe initializations without excess or deficiency.
Reliability R1.3	Pay attention to the range of the area pointed by a pointer.
Reliability R1.4	Use the object after constructing it completely.
Reliability R1.5	Pay attention to object creation and destruction.

28 Part 2 Coding Practices for Embedded Software: Practices Chart

ESCR C++

R1.1 Use areas after initializing them.

R1.1.1	Automatic variables shall be initialized at the time of declaration, or the initial values shall be assigned just before using them.	Preference guide	●
		Rule specification	

Compliant example

```
void func(){
  int var1 = 0; // Initialize at the time of
                // declaration
  int i;        // Do not initialize at the
                // time of declaration
  ...
  var1++;
  // Assign the initial value just before
  // using it
  for (i = 0; i < 10; i++) {
    ...
  }
}
```

Non-compliant example

```
void func(){
  int var1;
  ...
  var1++;
  ...
}
```

If automatic variables are not initialized, their values become undefined and the operation results may differ depending on behavior and environment. The initialization must be at the time of declaration or the initial values must be assigned just before using them.

R1.2 Describe initializations without excess or deficiency.

R1.2.2	Initialization of enumeration type (enum type) members shall be by either: not specifying any constants; specifying all the constants; or specifying only the first member.	Preference guide	
		Rule specification	

Compliant example

```
// A different value is assigned
// respectively from E1 to E4.
enum etag {E1 = 9, E2, E3, E4};
enum etag var1;
var1 = E3;
// E3 and E4 in var1 will never be equal.
if (var1 == E4)
```

Non-compliant example

```
// Both E3 and E4 become 11 unintentionally.
enum etag {E1, E2 = 10, E3, E4 = 11};
enum etag var1;
var1 = E3;
// It will be true despite the intention,
// because E3 and E4 are equal
if (var1 == E4)
```

*R1.1.2 and R1.2.1 are deleted in C++ Languageedition and are vacant. (see the table in Appendix)

Reliability R1 ● Initialize areas and use them by taking their sizes into consideration. 29

① Quality concept

Quality concepts are related to the main quality characteristics defined in "ISO/IEC9126-1." This guide uses the following four quality concepts:

Reliability Maintainability Portability Efficiency

② Practice

Describes the practice to be followed by programmers during coding

- In outline — Defines the general concept of the practice. It is not dependent on programming languages.
- In detail — Elaborates the general concept of the practice with more specific points that should be noted. Like practice in outline, it is basically programming language-independent, but some are stated as C++ language-specific.

③ Rule number

Identification number of each rule

④ Rule

Specific reference rule or rules for C++ language corresponding to the practice that must be followed

⑤ Preference guide

Provides supportive information (marks) to indicate whether the corresponding rule described under each practice should be chosen as a part of the newly created coding convention or not.

No mark	Rules considered to be appropriate to choose, based on the project characteristics.
●	Rules considered unnecessary to be included in the coding convention, when seen from the eyes of those who are very knowledgeable about the language specification (i.e.: rules that are already too common and obvious to experienced programmers).
○	Rules considered to significantly impair the quality characteristics if they are not followed.

⑥ Rule specification

Provides supportive information (verbal indicators) to indicate which rule need to be defined more specifically in detail or not, depending on the project policy, or should be prescribed in a document, such as, when it is recommended to "record the behavior and usage of compiler-dependent language specification as a document" (the latter is referred to as the "documentation rule" which can be used as it is, but is strongly recommended to be documented in more detail for various reasons).

No mark	Rules that do not need to be defined further in depth or prescribed in a document with more specific details
Choose	Rules required to be chosen from a list of multiple alternatives. Each alternative is numbered, using a parenthesized numeral (e.g.: (1), (2), ...).
Define	Specific rules that need to be defined for each project. The part to be defined is enclosed by《 》.
Document	Rules that need to be prescribed in a document. The part to be documented is enclosed by《 》.

⑦ Compliant example

Example of source code written in compliance with the rule

⑧ Non-compliant example

Example of source code violating the rule

⑨ Remarks

Provides notes pertaining to C++ language specifications, and explanation on why the particular rule is necessary and what kind of problem(s) may be caused by violation of that rule, among others.

Difference from C Language Edition

ESCR [C language edition] is used as the base document for creating this guide for C++ language. While the use of C++ language is beneficial for enhancing the reliability and maintainability of the written code and improving the productivity of programming through the extensive application of newly featured techniques like object-oriented programming and data abstraction, it is also considered desirable to mitigate the risks involved in the adoption of such new features by limiting the use of exceptions, templates and other newly introduced techniques that are difficult to apply successfully. To respond to these technical issues, many practices and rules that are mainly concerned with how classes, exceptions and templates written in C++ affect the program behavior have been added to this guide.

It is highly conceivable that, depending on the programming needs in a given project, there are often cases that require the flexible use of both C and C++ languages as the situation demands. This guide takes account of those who are already using ESCR [C language edition], and lists the practices and corresponding rules that are common to both C and C++ languages as much as possible in the same way presented in C language edition, including the use of same identification numbers. However, with respect to compliant and non-compliant examples, some have been changed to those that C++ programmers would feel more familiar. Furthermore, some rules that are based specifically on C++ language specifications that differ from C language specifications have been newly added, while there are also some rules that are numbered the same as in the C language edition, but have been revised to reflect the difference between C and C++ language specifications.

On the other hand, there are practice and rules that were provided in C language edition but have been omitted from this guide, which are as follows:

Practice: R2.2
Rules: R1.1.2, R1.2.1, R2.2.1, R2.8.1, R3.1.3, R3.1.4, M1.3.3, M4.4.4, M4.4.5

Terminology Used in the Practices Chart

The meaning of the terms used in the chart is as respectively explained in the table below:

Term	Meaning
Access	Reference to variables or the reference with modification.
Type specifier	Specifies a data type. There are two type specifiers, one that specifies basic types such as `char`, `int` and `float` and the other that specifies types uniquely defined with `typedef`s by the programmer for their own.
Type qualifier	Adds specific attributes to types. There are two type qualifiers: `const` and `volatile`.
Class type	Type defined by `class`, `struct` or `union`.
Storage class specifier	Specifies the location where data is stored or the range where data is applied. There are five specifiers: `register`, `static`, `extern`, `mutable`, and `thread_local`.
Boundary alignment	For example, if `int` type is 4 bytes, be sure to allocate such data from an address of the memory that is a multiple of four and never from an address that is not a quadruple.
Trigraph sequence	Defined sequences of three characters such as '`??=`', '`??/`', '`??(`' for the compiler to replace with single character. '`??=`', '`??/`', '`??(`' are interpreted into '`#`', '`\`', '`[`' respectively.
Lifetime	Duration that the access to a variable from the program is guaranteed after it is generated.
Multibyte character	A character expressed by data of two or more bytes. Chinese characters, Japanese characters, and Unicode characters are included.
Null pointer	A pointer that is not equivalent to any pointers that point to data or functions.
Null character	A character that expresses the end of a string. Expressed with '`\0`'.
Scope	The range in the program within which the identifier (such as, variable name) can be used. File scope shall be the scope within the given source file.
Side effect	Processing that causes changes to the state of execution environment. Side effect occurs when processing: reference and change to volatile data, change to data, change to files, and function-calls that perform these operations.
Block	A range that is enclosed with braces '{', '}' in data declarations and programs, etc.
Enumeration type	Type declared by `enum` or `enum class (enum struct)`. Constructed with several enumerated members.
Enumerator	Members of an enumerated type.
Polymorphic	Programming capability to operate data (objects) of various types using a common interface.

Coding Practices for Embedded Software

This part presents the coding practices for embedded software. As explained earlier, the practices are categorized according to the perspective of four characteristics (quality concepts): “reliability”, “maintainability”, “portability” and “efficiency”, which have been adopted from the software quality characteristics defined in ISO/IEC9126-1. Please note, however, that these practices have been categorized in this way basically for the sake of convenience of the users of this guide, and that there are actually some useful practices and corresponding rules that can be applied to improve more than one characteristic (e.g.: both reliability and maintainability).

Moreover, the coding practices respectively related to these quality characteristics and the reference rules that support the correct ways of executing these practices are also described in this part of the guide.

Reliability	R	Practices to improve the reliability of software that has been developed fall under this category. Main points taken into consideration include: - Minimizing problems arising while using the software; - Increasing tolerability against bugs and interface violation.
Maintainability	M	Practices to create source code that is easy to modify and maintain fall under this category. Main points taken into consideration include: - Making the code easy to understand and modify; - Minimizing the impact of modifications on the entire code; - Making the modified code easy to check.
Portability	P	Practices to port the software program that has been created on the assumption of being used to operate under a certain environment to another environment as efficiently as possible without error fall under this category.
Efficiency	E	Practices to effectively utilize the performance and resources of the software that has been developed fall under this category. Main points taken into consideration include: - Coding that is processing time-conscious; - Coding that takes account of memory size.

Reliability

A large number of embedded software is incorporated into products and used to support our daily lives in various situations. Consequently, the level of reliability demanded to quite a number of embedded software is extremely high. Software reliability requires the software to be capable of not behaving wrongly (not causing failure), not affecting the functionality of the entire software and system in case of malfunction, and promptly restoring its normal behavior after a malfunction occurs.

At the source code level, the point to be noted in regard to software reliability is the need of contriving methods to avoid coding that may cause such malfunctions as much as possible.

- **Reliability R1: Initialize areas and use them by taking their sizes into consideration.**
- **Reliability R2: Use data by taking their ranges, sizes and internal representations into consideration.**
- **Reliability R3: Write in a way that ensures intended behavior.**

Reliability R1 Initialize areas and use them by taking their sizes into consideration.

Various variables are used in programs written in C++ language. Without considering the areas to be reserved in the computer and ensuring that these areas are already initialized by the time these variables are used, unexpected malfunctions may occur.

Moreover, the pointers in C++ language need to be used carefully by being conscious of the areas they point to. Since the misuse of pointers may cause serious problems to the entire system, particular caution is necessary when using them.

Reliability R1.1	**Use areas after initializing them.**
Reliability R1.2	**Describe initializations without excess or deficiency.**
Reliability R1.3	**Pay attention to the range of the area pointed by a pointer.**
Reliability R1.4	**Use the object after constructing it completely.**
Reliability R1.5	**Pay attention to object creation and destruction.**

R1.1 Use areas after initializing them.

R1.1.1 Automatic variables shall be initialized at the time of declaration, or the initial values shall be assigned just before using them.

Preference guide	●
Rule specification	

Compliant example

```
void func(){
  int var1 = 0; // Initialize at the time of
                // declaration
  int i;        // Do not initialize at the
                // time of declaration
  ...
  var1++;
  // Assign the initial value just before
  // using it
  for (i = 0; i < 10; i++) {
    ...
  }
}
```

Non-compliant example

```
void func(){
  int var1;
  ...
  var1++;
  ...
}
```

If automatic variables are not initialized, their values become undefined and the operation results may differ depending on behavior and environment. The initialization must be at the time of declaration or the initial values must be assigned just before using them.

R1.2 Describe initializations without excess or deficiency.

R1.2.2 Initialization of enumeration type (enum type) members shall be by either: not specifying any constants; specifying all the constants; or specifying only the first member.

Preference guide	
Rule specification	

Compliant example

```
// A different value is assigned
// respectively from E1 to E4.
enum etag {E1 = 9, E2, E3, E4};
enum etag var1;
var1 = E3;
// E3 and E4 in var1 will never be equal.
if (var1 == E4)
```

Non-compliant example

```
// Both E3 and E4 become 11 unintentionally.
enum etag {E1, E2 = 10, E3, E4 = 11};
enum etag var1;
var1 = E3;
// It will be true despite the intention,
// because E3 and E4 are equal
if (var1 == E4)
```

*R1.1.2 and R1.2.1 are deleted in C++ Language edition and are vacant. (see the table in Appendix)

If an initial value is not specified to a member of the enumeration type, the value of the immediately preceding member plus 1 (the value of the first member is 0) will be specified to this member. If some initial values are specified while others are not, the same value may unintentionally be assigned to different members and may become the cause of unexpected behavior. To prevent the same value from being assigned to different members, initialization of the members must be by either not specifying any constants, specifying all the constants, or specifying only the first member, depending on the usage.

R1.3 Pay attention to the range of the area pointed by a pointer.

R1.3.1

(1) Integer addition to or subtraction from (including ++ and --) pointers shall not be made; Array format with [] shall be used for references and assignments to the allocated area.

(2) Integer addition to or subtraction from (including ++ and --) pointers shall be made only when the pointer points to the array and the result must be pointing within the range of the array.

Preference guide	●
Rule specification	choose

Compliant example

```
#define N 10
int data[N];
int *p;
int i;
p = data;
i = 1;

Compliant example of (1) and (2)
data[i] = 10;  // Compliant
data[i + 3] = 20;  // Compliant

Compliant example of (2)
*(p + 1) = 10;
```

Non-compliant example

```
#define N 10
int data[N];
int *p;
p = data;

Non-compliant example of (1)
*(p + 1) = 10;  // Non-compliant
p += 2;  // Non-compliant

Non-compliant example of (2)
*(p + 20) = 10;  // Non-compliant
```

Performing operations on pointers can blur the destinations pointed by the pointers. It raises the possibility of implanting bugs that is likely to refer or write to unsecured areas. Rather, using an array name that points to the beginning of the area and to access elements of the array with indices will make the program safer. A dynamic memory area obtained by malloc should be treated as an array, and a pointer to the starting address of the area should be handled as the array name.
For multi-dimensional array, this rule applies to each partial array.
Regarding rule (2), it is permissible to point to the area directly after the last element of the array as long as the array element is not accessed. In other words, in the case where `int data[N]` and `p=data`, `p+N` complies with the rule as long as it is not used for accessing the array elements, whereas, using, such as, `*(p+N)` that accesses an array element is non-compliant.

R1.3.2 Subtraction between pointers shall be used only when both pointers are pointing at elements in the same array.

Preference guide	●
Rule specification	

Compliant example

```
int off, var1[10];
int *p1, *p2;
p1 = &var1[5];
p2 = &var1[2];
off = p1 - p2;  // Compliant
```

Non-compliant example

```
int off, var1[10], var2[10];
int *p1, *p2;
p1 = &var1[5];
p2 = &var2[2];
off = p1 - p2;  // Non-compliant
```

By subtracting from one pointer to another pointer, the number of elements existing between the two elements pointed respectively by these pointers will be expressed. If each pointer used in the subtraction is pointing at a different array, the way the variables laid out between the two is implementation-dependent and the execution results are not guaranteed. This implies that subtraction between pointers is meaningful only when both pointers are pointing at elements in the same array. Therefore, before subtracting from one pointer to another pointer, the programmer must ensure that both pointers are pointing at the same array.

[Related rule] R1.3.3

R1.3.3 Comparing which pointer is greater or less than the other pointer shall be used only when two pointers are both pointing at either the elements in the same array, the members with the same access control defined in the same structure or class, or the members of the same structure.

Preference guide	●
Rule specification	

Compliant example

```
constexpr int N = 10;
char var1[N];
char* p = var1;

  ... // operations on p
if (p < var1+N) { ... } // Compliant
```

Non-compliant example

```
constexpr int N = 10;
char var1[N];
char var2[N];
char* p = var1;

  ... // operations on p
if (p < var2+N) { ... } // Non-Compliant
```

Comparing the addresses of different variables does not cause a compile error, but is meaningless because the address of the variable is implementation-dependent. In addition, the behavior of such comparison is not defined (undefined behavior).

[Related rules] R1.3.2 R2.7.3

R1.4 Use the object after constructing it completely.

R1.4.1

All the data members of a class shall be initialized in its constructor. The initialization procedure shall be as follows:

1. Initialization shall be written in the declaration of members that are always initialized with the same value. The constructor initializer shall be used to perform the initialization of other members. However, this shall not apply in case of initializing multiple non-class type members with the same value.
2. The constructor initializer shall have the base class and data members written in the order they are declared.
3. The constructor initializer shall not use any other data members of the same class for initialization. Or, if there is a need to use any of these other members, they shall be limited to only those declared before the specific data member targeted for initialization.

Preference guide	○
Rule specification	

Compliant example

```
class CLS
{ public:
    CLS(int x) : cls_i(x), cls_j(cls_i) {
   // Compliant: Initialized in the order of
   // declaration, starting from cls_i.
   // No members are used for initialization
  }
private:
  int cls_i;
  int cls_j;
};
class CLS  {
public:
  CLS(int x) : cls_i(x), cls_j(cls_i) {
   // Compliant: Initialized in the order of
   // declaration, starting from cls_i.
  }
private:
// Same as above: Omitted.
};

class CLS  {
public:
  CLS(int x) {
    cls_i = cls_j = x;
  }
private:
// Same as above: Omitted.
};
class CLS  {
public:
  CLS(int x) : cls_j(x) {
   // Compliant: Constructor initializer is
   // used to initialize cls_j, since its
   // value is determined at the time an
   // object is created.
    ...
  }
```

Non-compliant example

```
class CLS  {
public:
  CLS(int x) : cls_j(x), cls_i(cls_j) {
   // Non-compliant: Initialized from cls_j
   // without following
   // the order of declaration.
   // Because initialization is performed
   // from cls_i ,
   // cls_j is uninitialized at the time of
   // initialization of cls_i.
  }
private:
  int cls_i;
  int cls_j;
};
```

```
private:
    int cls_i = 0;
     // Compliant: Initialization is written
     // in member declaration, since cls_i
     // is always initialized with zero (0).
    int cls_j;
} ;
```

Explicit initialization of every member in the constructor (excluding static data members) can prevent all the members that need to be initialized from being left uninitialized.
For initialization, write the initialization in the declaration of the member or use the constructor initializer. (See also E1.1.6.) Writing the initialization in the declaration of the member is a method introduced in C++11 as a new feature. This method of initialization is easy to understand. In addition, this method has little risk of initialization errors when initializing the members of the same class type always with the same value. But if there is a need to initialize multiple members of a different class with the same value, initialize them through assignment in the constructor body. Doing so will improve the readability and mitigate the risk of initialization errors.
Moreover, write the members in the constructor initializer in the order they are declared. The constructor initializer initializes the members in the order they are declared in the class, and not in the order they are written. By writing the members in the initializer in the order they are declared, description to use an uninitialized variable, as shown in the non-compliant example above, can be prevented.
In C++11, the feature to describe the initialization in the data member declaration (with member initializer in the class) has been added. The use of this feature is recommended when the data member is always initialized with the same value. It is easy to understand and therefore will not be mistaken easily.

[Related rules] R1.4.3 R1.4.4 R1.4.5 M2.3.1 E1.1.6

R1.4.2 Non-static data members shall be all copied whenever a copy constructor or copy assignment operator is used.

Preference guide	○
Rule specification	

Compliant example

```
class Base {
public:
  Base() : base_i(0) { }
  Base(int x) : base_i(x) { }
  Base(const Base &r) : base_i(r.base_i) { }
  // Compliant: All the data members are
  // initialized.
private:
  int base_i;
};

class Derived : public Base {
public:
  Derived(int x, int y) :
    Base(x), derive_j(y)  { }
  Derived(const Derive &r) :
    Base(r), // Compliant: The copy
             //constructor of the base
             // class is called.
    derive_j(r.derive_j){ } // Compliant: Data
                            //members of the
                            //derived class are
                            //initialized.
private:
  int derive_j;
};
```

Non-compliant example

```
class Base {
public:
  Base() : base_i(0) { }
  Base(int x) : base_i(x) { }
  Base(const Base &r) : base_i(0){ }
     // Non-compliant: A value unrelated to
     // r is assigned.
private:
  int base_i;
};

class Derived : public Base {
public:
  Derived(int x, int y) :
    Base(x), derive_j(y)  { }
  Derived(const Derive &r) :
     // Non-compliant: Forgot to copy the
     // base class.
     // Base part is initialized.
    derive_j(r.derive_j){ }
private:
  int derive_j;
};
```

When the copy constructor or copy assignment operator is automatically generated by the compiler, all the data members (excluding the static data members) are copied. All these data members must also be copied if user-defined copy constructor or copy assignment operator is going to be used. When there are any data members that are not copied, the value of each data member that is not copied will become undefined and may become the cause of unintended behavior.

In case of derived class, all the data members in the base class must also be copied. Therefore, when a copy constructor is used, be sure to include and call the copy constructor of the base class. Likewise, when a copy assignment operator is used, be sure to include and call the copy assignment operator of the base class.

Useful information Functions generated automatically by the compiler

The compiler will generate the following functions automatically when they are not declared in the class:

- Default constructor (only when there is no constructor declaration);
- Copy constructor;
- Copy assignment operator;
- Destructor.

In C++11, the following functions are also generated automatically:

- Move constructor
- Move assignment operator

Example:

```
class C {
public:
    int m;
};
```

The above means the same as the description below:

```
class C {
public:
    int m;
    C() = default; // Default constructor
    ~C() { } = default; // Destructor
    C(const C &) = default; // Copy constructor
    C &operator = (const C &) = default; // Copy assignment operator
    C(C&&) = default; // Move constructor
    C& operator = (C&&) = default; // Move assignment operator
};
```

【Reference materials for those wanting to know more in detail about this rule】

- The C++ Programming Language [Fourth Edition]   17.6
- Effective Modern C++   Item 11 Item 17 Item 22

R1.4.3 Member function for only reading data members shall be called after the constructor initializes the object completely and before the destructor starts destroying the object.

Preference guide	●
Rule specification	

Compliant example

```
(1)
class CLS {
public:
  CLS() {
    init();  // Compliant: Performs only
             // write access.
  }
private:
  time_t cls_time;
  void init() {  // Initialization of the
                 // class is processed
    cls_time = time(0);
  }
};

(2)
class CLS {
public:
  CLS() {
    init();
    show();  // Compliant: Display after
             // completing the
             // initialization of the class
  }
  void show() {  // Display is processed.
    cout << "Current date and time:"
         << ctime(&cls_time) << endl;
  }
private:
  time_t cls_time;
  void init() {  // Initialization of the
                 // class is processed.
    cls_time = time(0);
  }
};
```

Non-compliant example

```
class CLS {
public:
  CLS() {
    show();  // Non-compliant: Display before
             // completing
             // the initialization of the
             // class.
    init();
  }
  void show() {  // Display is processed.
    cout << "Current date and time:"
         << ctime(&cls_time) << endl;
  }
private:
  time_t cls_time;
  void init() {  // Initialization of the
                 // class is processed.
    cls_time = time(0);
  }
};
```

When the member function for reading and referencing data members is called from the constructor or destructor, uninitialized or destroyed data member may be referenced.

[Related rules] R1.4.1 R1.4.5 R1.4.6

R1.4.4 Virtual function shall not be called from the constructor and destructor.

Preference guide	●
Rule specification	

Compliant example

```
class Base {
public:
  Base(int x = 1) : base_i(x) { }
                        // Compliant: Virtual
                        // function is
                        // not called
  int getI() {return base_i;}
private:
  int base_i;
};

class Derived : public Base {
public:
  Derived(int x) : Base(10), derive_j(x) { }
    // Base class is initialized with 10.
private:
  int derive_j;
};
```

Non-compliant example

```
class Base {
public:
  Base() {
    base_i = getInitValue();
                    // Non-compliant: Virtual
                    //function (1) is called
  }
  virtual int getInitValue() {
    // (1) Constructor of the base class
    // always executes this function
    return 1;
  };
  int getI() {return base_i;}
private:
  int base_i;
};

class Derived : public Base {
public:
  Derived(int x) : Base(), derive_j(x) { }
  virtual int getInitValue() override {
  // getInitValue is a virtual function but
  // is not
  // called from the constructor of the base
  // class.
  return 10;
  }
private:
  int derive_j;
};
```

Virtual function call from the constructor or destructor will not behave polymorphically. For example, even when the virtual function is called from the constructor of the base class, the function overridden in the derived class will not be called.

[Related rules] R1.4.1 R1.4.5 R1.4.6

R1.4.5 The constructor and copy assignment operator shall respond to unsuccessful object construction.

Preference guide	●
Rule specification	

Compliant example

```
class CLS {
public:
  CLS(int i) : cls_pi(0) {
    if (i == 0) {
      throw 0;  // Compliant: Checks and
                // throws exception
                // before memory allocation.
    }
    try {
      cls_pi = new int[i];
     ... // Constructor performs its steps for
         //construction. Some kind of
         //exception may arise.
    } catch(...) {
      if (!cls_pi) {
        delete cls_pi;
           // Compliant: Throws exception
           // after freeing the memory.
      }
      throw;
    }
  }
  …
private:
  int *cls_pi;
};
```

Non-compliant example

```
class CLS {
public:
  CLS(int i) : cls_pi(new int[i]) {
    … // Constructor performs its steps for
      // construction.
      // When exception arises, cls_pi is
      // not destroyed.
    if (i == 0) {
      throw 0;  // Non-compliant: cls_pi that
                // has been created is not
                // destroyed.
    }
  }
  …
private:
  int *cls_pi;
};
```

When an exception is thrown by the constructor or copy assignment operator, the object will be in an incomplete state with data members that are not all fully initialized yet. In addition, the destructor will not be called.

As a result, if there are some data members that have been initialized successfully and already have memory allocated, memory leak will occur, because the memory that is supposed to be freed by the destructor is not freed.

To prevent such problem from occurring, make the constructor or copy assignment operator catch the exception and free the memory that has been allocated.

[Related rules] R1.4.1 R1.4.3 R1.4.4

R1.4.6 Catch handler described in the constructor or destructor shall not reference data members of that class.

Preference guide	●
Rule specification	

Compliant example

```
class CLS {
public:
  CLS(int i) : cls_pi(0) {
    try {
      … // Constructor performs its steps
        // for construction.
      if (flag == 0) {throw;}
        cls_pi = new int[i];
        for (int j = 0; j < i; j++) {
          cls_pi[j] = j;  // Value is set.
      }
    } catch(...) {
      // Compliant: Non-static data members
      // are not referenced in the catch
      // block of the destructor.
    }
  }
  ~CLS() {
    try {
      delete[] cls_pi;
      cls_pi = 0;
    } catch(...) {
      // Compliant: Non-static data members
      // are not referenced in the catch
      // block of the destructor.
    }
  }
private:
  int *cls_pi;
  bool flag;
};
```

Non-compliant example

```
class CLS {
public:
  CLS(int i) : cls_pi(0) {
    try {
      … // Constructor performs its steps
        // for construction.
      if (flag == 0) {throw;}
        cls_pi = new int[i];
        for (int j = 0; j < i; j++) {
          cls_pi[j] = j;  // Value is set.
      }
    } catch(...) {
      if (cls_pi[0] == 0) {
                // Non-compliant: Possible
                // that the value of
                // cls_pi is not set.
        return;
      }
    }
  }
  ~CLS() {
    try {
      delete[] cls_pi;
    } catch(...) {
      if (cls_pi[0] == 0) {
                // Non-compliant: cls_pi
                // is already destroyed.
        return;
      }
    }
  }
private:
  int *cls_pi;
  bool flag;
};
```

When data members are referenced by a catch handler described in the constructor or destructor, the catch handler may reference uninitialized or destroyed data member.

[Related rules] R1.4.3 R1.4.4

R1.5 Pay attention to object creation and destruction.

R1.5.1 The same form (whether with or without []) shall be used for new and corresponding delete.

Preference guide	●
Rule specification	

Compliant example

```
int *p1 = new CLS[10] :  // CLS is class type.
int *p2 = new CLS;
…
delete[] p1;  // Compliant: Freed in same
              // form as new.
delete p2;    // Compliant: Freed in same
              // form as new.
```

Non-compliant example

```
int *p1 = new CLS[10] :  // CLS is class type.
int *p2 = new CLS;
…
delete p1;      // Non-compliant: Freed in
                // different form from new.
delete[] p2;    // Non-compliant: Freed in
                // different form from new
```

The behavior is undefined when memory allocated with `new[]` is freed with `delete`. For example, problems like not all the allocated memory being freed, and destructor not being called for the number of elements in the allocated array may occur. Likewise, the behavior is undefined when memory allocated with `new` is freed with `delete[]`.

Therefore, to free the memory allocated with `new[]`, be sure to free it with `delete[]`. Likewise, to free the memory allocated with `new`, be sure to free it with `delete`.

Reliability R2 Use data by taking their ranges, sizes and internal representations into consideration.

The data used in programs vary in how they are represented internally and in the range they can be operated, depending on their types. When using these different types of data for operation, they must be written carefully by paying attention to various aspects, including precision and size. Otherwise, unexpected malfunctions may occur when they are processed in, such as, arithmetic operations. Therefore, there is a need to handle data with care, by taking their ranges, sizes and internal representations, among others, into consideration.

Reliability R2.1	**Make comparisons that do not depend on internal representations.**
Reliability R2.3	**Use the same data type to perform operations or comparisons.**
Reliability R2.4	**Describe code by taking operation precision into consideration.**
Reliability R2.5	**Do not use operations that have the risk of information loss.**
Reliability R2.6	**Use types that can represent the target data.**
Reliability R2.7	**Pay attention to pointer types.**
Reliability R2.8	**Write in a way that will enable the compiler to check that there are no conflicting declarations, usages and definitions.**

* Reliability R2.2 defined in C Language Edition has been omitted in C++ Language Edition, based on the judgment that it is not necessary to be defined as a coding practice for C++.

R2.1 Make comparisons that do not depend on internal representations.

R2.1.1 Floating-point expressions shall not be used to perform equality or inequality comparisons.

Preference guide	●
Rule specification	

Compliant example

```
#define LIMIT 1.0e-4
void func(double d1, double d2) {
  double diff = d1 - d2;
  if ((-LIMIT <= diff) && (diff <= LIMIT)) {
   …
Or
void func(double d1, double d2) {
  if (fabs (d1 - d2) <=
       std::numeric limits<double>::
       epsilon()) {
       …
```

Non-compliant example

```
void func(double d1, double d2) {
  if (d1 == d2) {
    …
```

In case of a floating-point type, values written in the source code do not exactly match with those actually implemented. Therefore, the comparison results must be judged by taking account of tolerance.

[Related rule] R2.1.2

R2.1.2 Floating-point variable shall not be used as a loop counter.

Preference guide	●
Rule specification	

Compliant example

```
void func() {
  int i;
  for (i = 0; i < 10; i++) {
    …
```

Non-compliant example

```
void func() {
  double d;
  for (d = 0.0; d < 1.0; d += 0.1) {
    …
```

If operations are repeatedly performed to a floating-point variable used as a loop counter, the intended result may not be achieved due to accumulated calculation errors. Therefore, for loop counter, use integer type variable.

[Related rule] R2.1.1

*R2.2 and R2.2.1 are deleted in C++ Language edition and are vacant. (see the table in Appendix)

R2.1.3 memcmp shall not be used to compare class-type objects.

Preference guide	●
Rule specification	

Compliant example

```
struct TAG {
  char c;
  long l;
};
TAG var1, var2;
void func() {
  if ((var1.c == var2.c) &&
       (var1.l == var2.l)) { // Compliant
     …
}
class CLS {
private;
  char c;
  long l;
public:
  …
  friend bool operator ==
    (const CLS &lhs, const CLS &rhs) const;
  friend bool operator !=
    (const CLS &lhs, const CLS &rhs) const;
};
CLS var3, var4;
bool operator ==
  (const CLS &lhs, const CLS &rhs) const {
  return (lhs.c == rhs.c) && (lhs.l == rhs.l);
}
bool operator !=
  (const CLS &lhs, const CLS &rhs) const {
  return !(lhs == rhs);
}
void func2() {
  if (var3 == var4) // Compliant
    …
}
```

Non-compliant example

```
struct TAG {
  char c;
  long l;
};
TAG var1, var2;
void func() {
  if (memcmp(&var1, &var2, sizeof(var1))
       == 0) { // Non-compliant
   …
}
class CLS {
private;
  char c;
  long l;
public:
  …
};
CLS var3, var4;
void func2() {
  if (memcmp(&var3, &var4, sizeof(var3))
       == 0) { // Non-compliant
    …
}
```

Class-type objects may include unused areas. Since it is not clear what may be in any of these unused areas, do not use memcmp for comparison of class-type objects.
When making comparisons, thoroughly compare the members corresponding with each other by using operator overload.

[Related rules] R1.6.2

R2.3 Use the same data type to perform operations and comparisons.

R2.3.1 Unsigned integer constant expressions shall be described within the range that can be represented with the result type.

Preference guide	
Rule specification	

Compliant example

```
#defne M 0xffffUL
if ((M + 1) > M)
// If long is 32 bits, there is no problem
// even when the number of bits of int is
// not 32.
```

Non-compliant example

```
#define M 0xffffU
if ((M + 1) > M)
// The result varies depending on whether the
// int is 16 bits or 32 bits.
// If int is16 bits, the operation result
// will wrap around and the comparison result
// will be false.
// If int is 32 bits, the operation result
// will be within the range of int and the
// comparison result will be true.
```

Unsigned integer operations in C++ language wrap around without overflow (the result will be the remainder of the maximum representable value). Because the overflow is not flagged, there is a risk of not noticing when the operation result differs from the intended result. For example, when there are two environments that differ in the number of bits of int, the same constant expression produces different operation results, depending on whether they exceed the representable value range or not.

R2.3.2 When using conditional operator (?: operator), the logical expression shall be enclosed in parentheses () and both return values shall be the same type.

Preference guide	
Rule specification	

Compliant example

```
void func(int i1, int i2, long e1) {
  i1 = (i1 > 10) ? i2 : static_cast<int>(e1);
```

Non-compliant example

```
void func(int i1, int i2, long e1) {
  i1 = (i1 > 10) ? i2 : e1;
```

When writing code using different types, perform a cast to explicitly state the intended type.

[Related rule] R1.4.1

R2.3.3 Loop counters and variables used for comparison of loop iteration conditions shall be the same type.

Preference guide	●
Rule specification	

Compliant example

```
void func(int arg) {
   int i;
   for (i = 0; i < arg; i++) {
```

Non-compliant example

```
void func(int arg) {
   unsigned char i;
   for (i = 0; i < arg; i++) {
```

Using comparison between variables with different ranges of representable values as a loop iteration condition may produce unintended results and end up in an infinite loop.

R2.4 Describe code by taking operation precision into consideration.

R2.4.1 When the type of an operation and the type of the destination to which the operation result is assigned (assignment destination) are different, the operation shall be performed after casting them to the type of expected operation precision.

Preference guide	●
Rule specification	

Compliant example

```
int i1, i2;
long l;
double d;
void func() {
  d = static_cast<double>(i1) /
      static_cast<double>(i2); // Divide using
                               // floating-
                               // point type.
  l = static_cast<long>(i1) << i2; // Shift
                                   // using
                                   // long.
```

Non-compliant example

```
int i1, i2;
long l;
double d;
void func() {
  d = i1 / i2;  // Divide using integer type.
  l = i1 << i2; // Shift using int.
```

The type used in operation is determined by the type of the expression (operand) used for the operation, and the type of the assignment destination is not taken into consideration at compile time. Therefore, do not expect the operation to output its result in the type of the assignment destination if the operating type differs from the destination type. When there is a need to execute an operation in the type that differs from the operand type, perform a cast to convert the type of operand to the intended type before operation.

[Related rule] R2.5.1

R2.4.2 When performing arithmetic operations or comparisons of expressions mixed with signed and unsigned, an explicit cast to the expected type shall be performed.

Preference guide	
Rule specification	

Compliant example

```
long l;
unsigned int ui;
void func() {
  l = l / static_cast<long>(ui);
  Or
  l = static_cast<unsigned int>(l) / ui;
  if (l < static_cast<long>(ui)) {
  Or
  if (static_cast<unsigned int>(l)< ui) {
```

Non-compliant example

```
long l;
unsigned int ui;
void func() {
  l = l / ui;
  if (l < ui) {
    …
```

Some operations, such as, size comparison, multiplication and division output different results, depending on whether they are performed with signed or unsigned. If an operation is written for a mixture of signedness, unsigned operation is not always executed because it is the compiler that determines which type to execute the operation in (whether with signed or unsigned) by taking account of the respective data sizes. Therefore, when performing an arithmetic operation of mixed signedness, there is a need to check whether the intended operation is with signed or unsigned, and perform an explicit cast to change the operating type to the desired type before operation so that the intended operation result can be expected.

Note: If there are data types that may have to be changed for use in intended operation, it is often better to change them rather than performing a cast mechanically. Therefore, in such a situation, first consider changing the data type.

R2.5 Do not use operations that have the risk of information loss.

R2.5.1

When performing assignments (=operation, actual arguments passing of function calls, function return) or operations to data types that may cause information loss, they shall be first confirmed that there are no problems, and a cast shall be described to explicitly state that they are problem-free.

Preference guide	○
Rule specification	

Compliant example

```
Assignment example
short s;  // 16 bits
long l;   // 32 bits
void func() {
  s = static_cast<short>(l);
  s = static_cast<short>(s + 1);
}
Operation example
unsigned int var1, var2; // int size is 16
                         // bits
var1 = 0x8000;
var2 = 0x8000;
if (static_cast<long>(var1) + var2 > 0xffff){
// The result is true.
```

Non-compliant example

```
Assignment example
short s;  // 16 bits
long l;   // 32 bits
void func() {
  s = l;
  s = s + 1;
}
Operation example
unsigned int var1, var2; // int size is 16
                         // bits
var1 = 0x8000;
var2 = 0x8000;
if (var1 + var2 > 0xffff) {
// The result is false
```

When a value is assigned to a variable that differs in type, the value may change (i.e. information may be lost). The assignment destination, therefore, should be the same type whenever possible. When a value is assigned to a different type intentionally in cases, such as, where there is no risk of information loss or no impact even if information is lost, perform a cast to explicitly state the intention.

In case of operations where the result exceeds the representable value range of the type used, the result may become an unintended value. Therefore, for safety, verify that the operation result is within the representable value range of the type used, or convert it to the type that could adequately accommodate larger values before operation.

Note: If there are data types that may have to be changed for use in intended operation, it is often better to change them rather than performing a cast mechanically. Therefore, in such a situation, first consider changing the data type.

[Related rule] R2.4.1

R2.5.2 Unary operator '-' shall not be used in unsigned expressions.

Preference guide	●
Rule specification	

Compliant example

```
int i;
void func() {
  i = -i;
```

Non-compliant example

```
unsigned int ui;
void func() {
  ui = -ui;
```

If a unary operator '-' is used in unsigned expression and the operation result falls out of representable value range of the original unsigned type, unintended behavior may occur.
For example, writing "if (-ui<0)" in the non-compliant example will not make this "if" true.

R2.5.3 When one's complement (~) or left shift (<<) is applied to unsigned char or unsigned short type data, an explicit cast to the type of the operation result shall be performed.

Preference guide	○
Rule specification	

Compliant example

```
unsigned char uc;
void func() {
  uc = static_cast<unsigned char>(~uc) >> 1;
```

Non-compliant example

```
unsigned char uc;
void func() {
  uc = (~uc) >> 1;
```

The result of operation using unsigned char or unsigned short type will be signed int type. When the sign bit turns on due to operation, the intended result may not be achieved. This is why casting to the type of the intended operation is necessary.

[Related rule] R2.5.4

R2.5.4 The right-hand side of a shift operator shall be zero or more, and less than the bit width of the left-hand side.

Preference guide	●
Rule specification	

Compliant example

```
unsigned char a;    // 8 bits
unsigned short b;   // 16 bits
b = static_cast<unsigned short>(a) << 12;
                // Clearly indicated that the
                // operation is 16 bits.
```

Non-compliant example

```
unsigned char a;    // 8 bits.
unsigned short b;   // 16 bits.
b = a << 12;        // There may be an error
                    // in the shift count.
```

The behavior of a shift operator whose right-hand side (shift count) specifies a negative value or a value equal to or larger than the bit width* at the left-hand side (value to be shifted) is not defined in C++ language standard and will vary depending on the compiler used. (* This bit width will be that of int type if the size is smaller than int.)
The intention of specifying a value up to the bit width of int type as the shift count, in case the left-hand side (value to be shifted) is of a type that is smaller in size than int is not comprehended easily by others, even though its behavior is defined in the language standard.

[Related rule] R2.5.3

R2.6 Use types that can represent the target data.

R2.6.1

(1) The types used for bit field shall only be `signed int` or `unsigned int`. If a bit field of 1 bit width is required, `unsigned int` type shall be used, and not the `signed int` type.

(2) Bool type, integer type that specified the signedness (`signed` or `unsigned`) or signedness-specified `enum` type shall be used for bit field. If a bit field of 1 bit width is required, the integer type that specified unsigned or `bool` type shall be specified.

(3) Integer type that specified the signedness (`signed` or `unsigned`), `bool` type or `enum` type shall be used for bit field. If a bit field of 1 bit width is required, the integer type that specified `unsigned` or `bool` type shall be specified.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1)
struct S {
  signed   int m1:2; //Compliant : (1)(2)(3)
  unsigned int m2:1  //Compliant : (1)(2)(3)
  unsigned int m3:4; //Compliant : (1)(2)(3)
};

Compliant example of (2)
struct S {
   unsigned   char m1:2; //Compliant : (2)(3)
   enum class ec { EcA, EcB, EcC } m2:2;
                         //Compliant : (2)(3)
   bool m3:1;            //Compliant : (2)(3)
};

Compliant example of (3)
struct S {
   enum e { EA, EB, EC } m1:2; //OK : (3)
};
```

Non-compliant example

```
Non-compliant example of (1)
struct S {
  int m1:2;        //Non-compliant: (1)(2)(3)
                   //int type that does not specify
                   //the signedness is used.
  signed int m2:1; //Non-compliant: (1)(2)(3)
                   //signed int type of 1 bit
                   //width is used.
  unsigned char m3:4;  //Non-compliant: (1)
                       //char type is used.
  enum e { EA, EB, EC } m4:2;
              //Non-compliant: (1)(2)
              //enum type that has not defined the
              //handling of signedness is used.
  bool  m5:1;      //Non-compliant: (1)
                   //bool type is used.
};

Non-compliant example of (2)
struct S {
  int m1:2;        //Non-compliant: (1)(2)(3)
                   //int type that does not specify
                   //the signedness is used.
  signed int m2:1;  //Non-compliant: (1)(2)(3)
                    //signed int type of 1 bit
                    //width is used.
  enum e { EA, EB, EC } m3:2;
              //Non-compliant: (1)(2)
              //enum type that has not defined the
              //handling of signedness is used.
};
Non-compliant example of (3)
struct S {
  int m1:2;        //Non-compliant: (1)(2)(3)
                   //int type that does not specify
                   //the signedness is used.
  signed int m2:1; //Non-compliant: (1)(2)(3)
                   //signed int type of 1 bit
                   //width is used.
};
```

(1) To be compatible with C90, use only the int type defined in C90, which is the "int type that has specified the signedness to either signed or unsigned". Do not use the "signedness-unspecified int type" that may become signed or unsigned, depending on the compiler used.

(2) Use either one of the following types defined in C++ language:
- Signedness-specified integer type
- `bool` type
- Enumeration types added in C++11 (`enum class`, `enum struct`, or `enum` *tag_name*`:` *type*)

Do not use the "signedness-unspecified integer type" that may become `signed` or `unsigned`, depending on the compiler used. Also, do not use the enumeration type that has not defined the handling of signedness.

(3) Do not use the "signedness-unspecified integer type" that may become `signed` or `unsigned`, depending on the compiler used.
- Signedness-specified integer type
- `bool` type
- Enumeration type

In C++ language, types like `char` and `short` can also be used for bit field. But note that their use is prohibited in (1). This rule is assumed to be applied in projects where both C language and C++ language are commonly used. In C90, `int` is the only type that can be specified as the bit field, and the signedness of the bit field of "signedness-unspecified `int` type" is implementation-dependent.

In C++ language, the signedness of the bit field of "signedness-unspecified `int` type" is implementation-dependent. Therefore, when integer type is used for bit field, the signedness must be specified. Moreover, while the signedness of enumeration types added in C++11 are specified (`enum class`, `enum struct`, or `enum` *tag_name*`:` *type*), the signedness of enumeration types defined in C++03 is implementation-dependent. Therefore, in (2), the use of enumeration type defined in C++03 is prohibited.
For the integer type bit field of 1 bit width, be sure to specify `unsigned`, since the only values that can be represented by `signed` integer type of 1 bit are '-1' and '0'.

[Related rule] P1.3.3

R2.6.2 Data used as bit sequences shall be defined with unsigned type, and not with the signed type.

Preference guide	
Rule specification	

Compliant example

```
unsigned int flags;
void set_x_on() {
  flags |= 0x01;
```

Non-compliant example

```
signed int flags;
void set_x_on() {
  flags |= 0x01;
```

The result of bitwise operation (~ , << , >> , & , ^ , |) to signed type may vary, depending on the compiler used.

R2.7 Pay attention to pointer types.

R2.7.1

(1) Pointer type shall not be converted to other pointer type or to integer type, and vice versa, with the exception of the following cases:
- Conversion from pointer to data type to `void*` type;
- Conversion between pointers to class type with base-derived relationship.

(2) Pointer type shall not be converted to other pointer type or to integer type with less data width than that of the pointer type, with the exception of the following cases:
- Mutual conversion between `void*` types in pointer to data type;
- Conversion between pointers to class type with base-derived relationship.

(3) Pointer to data type can be converted to pointer to other data type or to void* type, but pointer to function type shall not be converted to pointer to other function type or to pointer to data type. In case of converting pointer type to integer type, such conversion shall not be performed if the data width of the integer type is less than that of the pointer type.

Preference guide	○
Rule specification	Choose

Compliant example

```
int *ip;
int (*fp)(void) ;
char *cp;
int i;
void *vp;

Compliant example of (1)
ip = static_cast<int*>(vp);

Compliant example of (2)
i = reinterpret_cast<int>(ip);

Compliant example of (3)
i = reinterpret_cast<int>(fp);
cp = reinterpret_cast<char*>(ip);
```

Non-compliant example

```
int *ip;
int (*fp)(void) ;
char c;
char *cp;

Non-compliant example of (1)
ip = reinterpret_cast<int*>(cp);

Non-compliant example of (2)
c = reinterpret_cast<char>(ip);

Non-compliant example of (3)
ip = reinterpret_cast<int*>(fp);
```

If a pointer type variable is casted or assigned to another pointer type, it is difficult to identify what kind of data is contained in the area pointed by the pointer. With some MPUs, runtime errors occur if the destination of a pointer that is not at word boundary is accessed as `int` type; thus changing pointer types involves the risk of causing unexpected bugs. It is safer not to cast or assign pointer type variables to other pointer types. Converting pointer types to integral types is also risky, involving the same problem stated above. Such conversions, therefore, should be reviewed with experts, whenever deemed necessary. Moreover, attention must also be given to the value ranges of `int` type and pointer type. Be sure to check the specifications of the compiler beforehand, because there may be cases where the size of the pointer type is 64 bits even though the size of `int` type is 32 bits.
However, for conversion between pointers to class type with base-derived relationship, choose one of the rules described under R2.7.4 (and also refer to the summary below).

`<cstdint>` defines `intptr_t` and `uintptr_t`, which respectively represents `signed` and `unsigned` integer types with data width capable of holding a value converted from a pointer type and be converted back to that type with a value that equals to the original pointer. These types should be used when converting between pointer type and integer type.

[Related rules] R2.7.4 M4.1.2

Rules on pointer conversion

As explained in rules R2.7.1 and R2.7.4, conversion (assignment) of pointer type variables to other pointer types may output unintended result. Therefore, it is a risk to use such description more than is necessary. See below for the summary of rules on pointer conversion organized in tabular form.

In these tables, the columns show the pointer types converted (assigned) from, and the rows show the pointer types converted (assigned) to.

○ : Acceptable conversion

△ : Conversion is acceptable when converting from derived class to its base class, or from base class using dynamic_cast operator to its derived class.

× : Unacceptable conversion

R2.7.1 Rule (1)

Converted to / Converted from			Pointer to data		Pointer to function	Pointer to void	Integer type
			Pointer to class type	Pointer to other than class type			
Pointer to data	Pointer to class type	With base-derived relationship	△	×	×	○	×
		Without base-derived relationship	×				
	Pointer to other than class type		×	×			
Pointer to function			×		×	×	×
Pointer to void			×		×	—	×

R2.7.1 Rule (2)

Converted to / Converted from			Pointer to data		Pointer to function	Pointer to void	Integer type	
			Pointer to class type	Pointer to other than class type			Whose size is less than the data width of the pointer type	Whose size is equal to or more than the data width of the pointer type
Pointer to data	Pointer to class type	With base-derived relationship	△	×	×	○	×	○
		Without base-derived relationship	×					
	Pointer to other than class type		×	×				
Pointer to function			×		×	×	×	○
Pointer to void			○		×	—	×	○

R2.7.1 Rule (3)

Converted to / Converted from			Pointer to data		Pointer to function	Pointer to void	Integer type	
			Pointer to class type	Pointer to other than class type			Whose size is less than the data width of the pointer type	Whose size is equal to or more than the data width of the pointer type
Pointer to data	Pointer to class type	With base-derived relationship	○	○	×	○	×	○
		Without base-derived relationship	○					
	Pointer to other than class type		○	○				
Pointer to function			×		×	×	×	○
Pointer to void			○		×	—	×	○

R2.7.2 A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

Preference guide	○
Rule specification	Choose

Compliant example

```
void func(const char *);
const char *str;
void x() {
  func(str);
   …
}
```

Non-compliant example

```
void func(char *);
const char *str;
void x() {
  func(const_cast<char*>(str));
   …
}
```

Be careful when accessing the areas qualified by `const` or `volatile`, because they are only for reference and must not be optimized. If a cast that removes any `const` or `volatile` qualification from the type addressed by a pointer is performed, the compiler will not be able to check and detect error descriptions in the program even if there are any, or may perform an unintended optimization.

R2.7.3 Comparison to check whether a pointer is negative or not shall not be performed.

Preference guide	●
Rule specification	

Compliant example

—

Non-compliant example

```
int *func1() {
  …
  return reinterpret_cast<int*>(-1);
}

int func2() {
  …
  if (func1() < 0) {
       // Comparison intended to check whether
       // the pointer is negative or not.
    …
  }
  return 0;
}
```

It is meaningless to compare whether a pointer is larger or smaller than 0.
When the subject of comparison is a pointer, the compiler will convert 0 into a null pointer. Therefore, even when the comparison of pointer against 0 is intended, the comparison will actually be between two pointers, and the intended behavior may not be achieved.

[Related rule] R1.3.3

R2.7.4

(1) A pointer to a derived class may be converted to a pointer to its base class. However, a pointer to the base class may not be converted to a pointer to its derived class.

(2) A pointer to a derived class may be converted to a pointer to its base class. Moreover, if `dynamic_cast` operator is used, a pointer to a base class may also be converted to a pointer to its derived class.

Preference guide	○
Rule specification	Choose

Compliant example

```
class Base { … };
class Derive1 : public Base { … };
Derive1 d;

Compliant example of (1)
// Compliant: Pointer to a derived class is
// converted to pointer to its base class.
Base *bp =&d;

Non-compliant example of (2)
Derive1 *dp;
// Compliant: dynamic_cast is used to cast
// to the pointer of the class with
// inheritance relationship
dp = dynamic_cast<Derive1*>(bp);
```

Non-compliant example

```
class Base { … };
class Derive1 : public Base { … };
class Derive2 : public Base { … };

Non-compliant example of (1) (2)
Derive1 *bp;
Derive2 *dp;
dp = dynamic_cast<Derive2*>(bp);
// dynamic_cast is used to cast, but because
// the object pointed by bp is Derive1 type,
// runtime error occurs
// when it is converted to Derive2 type
// pointer.
```

This rule pertains to conversion between pointers to class type with base/derived relationship.
The behavior is undefined when the type conversion from base class to derived class is performed without using `dynamic_cast` operator. Therefore, when it is unavoidable to convert a pointer to a base class to a pointer to its derived class, be sure to perform the conversion by using `dynamic_cast` operator, and check that the conversion has been successful.

[Related rules] R2.7.1 M4.1.2

*R2.8.1 is deleted in C++ Language edition and are vacant. (see the table in Appendix)

R2.8 Write in a way that will enable the compiler to check that there are no conflicting declarations, usages and definitions.

R2.8.2

(1) Functions shall not be defined with a variable number of arguments.

(2) When using functions with a variable number of arguments,《they shall be used after documenting the intended behaviors based on the compiler used》.

Preference guide	
Rule specification	Choose / Document

Compliant example

```
Compliant example of (1)
int func(int a, char b);
```

Non-compliant example

```
Non-compliant example of (1)
int func(int a, char b, ... );
```

Without understanding the behavior of functions with a variable number of arguments in the processing system, their use may cause stack overflow or other unexpected results.
In addition, when the number of arguments is variable, the number and the types of the arguments are not explicitly specified, and it will lower the readability of the code.

[Related rule] R2.8.3

R2.8.3 One prototype declaration shall be made at one location, so that it will be referenced by both its function calls and function definition.

Preference guide	○
Rule specification	

Compliant example

```
-- file1.h --
float f(int i);  // Compliant: Described in
                 // one location

-- file1.cpp --
#include "file1.h"
float f(int i) {
  …
}

-- file2.cpp --
#include "file1.h"
void g(void) {
  float x = f(1);
  …
}
```

Non-compliant example

```
-- file1.cpp --
float f(int i);  // Non-compliant: Declared
                 // in multiple files.
float f(int i) {
  …
}

-- file2.cpp --
float f(int i);  // Non-compliant: Declared
                 // in multiple files
                 // Mistaken with float f(int i);.
void g(void) {
  float x = f(1);
  …
}
```

This rule is for preventing the prototype declaration and function definition from being inconsistent. In case of overloading in C++ language, declaring a different return value type to a different file will not cause a compile error even when the description of the argument is the same, as shown in the above non-compliant example. That is because the return type will be ignored. However, such declaration may lead to unintended behaviors.

[Related rules] R2.8.2

Reliability R3 Write in a way that ensures intended behavior.

It is essential to be thorough with describing how to handle all the potential errors, by also taking account of unexpected events that may occur in cases that are even conceived as highly unlikely from the standpoint of program specifications. Moreover, writing code in ways that do not rely on language specifications, such as, explicit indication of operator precedence can also improve safety. To achieve high reliability, it is desirable to make every effort to avoid coding that leads to malfunction and write in a way that ensures intended behavior and safety as much as possible.

Reliability R3.1	Write in a way that is conscious of area size.
Reliability R3.2	Prevent operations that may cause runtime error from falling into error cases.
Reliability R3.3	Check the interface restrictions when a function is called.
Reliability R3.4	Do not perform recursive calls.
Reliability R3.5	Pay attention to branch conditions and describe how to handle cases that do not follow the predefined conditions when they occur.
Reliability R3.6	Pay attention to the order of evaluation.
Reliability R3.7	Pay attention to the behavior of classes.
Reliability R3.8	Pay attention to the behavior of exceptions.
Reliability R3.9	Pay attention to the behavior of templates.
Reliability R3.10	Pay attention to the behavior of lambda expressions.
Reliability R3.11	Be careful with how to access the shared data in programs that use threads or signals.

R3.1 Write in a way that is conscious of area size.

R3.1.1

(1) In an extern declaration of an array, the number of elements shall always be specified.

(2) In an extern declaration of an array, the number of elements shall always be specified, except for extern declarations of arrays that correspond to the array definition that includes initialization and has omitted the number of elements.

Preference guide	○
Rule specification	Choose

Compliant example

```
Compliant example of (1)
extern char *mes[3];
…
char *mes[] = {"abc", "def", nullptr};

Compliant example of (2)
extern char *mes[];
…
char *mes[] = {"abc", "def", nullptr};

Compliant example of (1), (2)
extern int var1[MAX];
…
int var1[MAX];
```

Non-compliant example

```
Non-compliant example of (1)
extern char *mes[];
…
char *mes[] = {"abc", "def", nullptr};

Non-compliant example of (1), (2)
extern int var1[];
…
int var1[MAX];
```

Making an `extern` declaration without specifying the size of the array will not cause error. However, omission of the size may cause problems in checking outside of the array range. Therefore, make it a rule to always specify the array size in the declaration.
However, there are exceptional cases where the array size can be omitted in the declaration, such as, when the array size is underspecified until it is determined by the number of initial values.
In C++11, a library for handling arrays and pointers safely is defined as standard specification. For example, in `std::array` class, using member function `at` to reference the array elements makes it possible to detect whether the reference is out of scope or not. The use of such library is recommended when the use of exceptions does not go against the project policy.

[Related rule] R3.1.2

R3.1.2 For a loop to sequentially access array elements, its iteration conditions shall include the judgment on whether the access is within the range of the array or not. However, for a loop to sequentially access array elements from the beginning of the array, range-based `for` loop shall be used.

Preference guide	
Rule specification	

Compliant example

```
char v1[MAX];
for (int i = FIRST; i < MAX && v1[i] != 0; i++){
// Compliant: Even if 0s are not specified in
// the var1 array, there is no risk of
// accessing outside the array range.
  …

//Judgement is made by using size().
vector<char> v2;
for (int i = FIRST; i < v2.size() && v2[i]!=0; i++) {
// Compliant: Judge the range by size().
  …

// When using range-based for loop
int array[5] = { 1, 2, 3, 4, 5 };
for (char x : v1)
    std::cout << x << std::endl;
// Compliant: 1 2 3 4 5 will be displayed
// in standard output.
```

Non-compliant example

```
char v1[MAX];
for (int i = 0; v1[i] != 0; i++) {
// Non-Compliant: If 0s are not specified in
// the var1 array, there is a risk of
// accessing outside the array range.
  …

vector<char> v2;
for (int i = 0; v2[i] != 0; i++) {
// Non-Compliant: No Judgment made on the
// range
  …
```

This rule is to prevent accessing outside the array range, which is a bug that frequently occurs in C and C++ languages and can often lead to a serious problem. By using the range-based for statement introduced in C++11, access to outside the array range can be prevented.

The C++ Programming Language, Fourth Edition also advises that the use of range-based `for` statement should be prioritized over for statement, if available. (Chapter 9)

[Related rule] R3.1.1

*R3.1.3 and R3.1.4 are deleted in C++ Language edition and are vacant. (see the table in Appendix)

R3.2 Prevent operations that may cause runtime error from falling into error cases.

R3.2.1 Operations shall be performed after confirming that the right-hand side expression of division or remainder operation is not 0.

Preference guide	
Rule specification	

Compliant example

```
if (y != 0)
    ans = x / y;
```

Non-compliant example

```
ans = x / y;
```

Apart from when the value is obviously not 0, the right-hand side expression of division or remainder operation must be confirmed that it is not 0 before performing the operation. Otherwise, division by zero error may occur at runtime.

[Related rules] R3.2.2 R3.3.1

R3.2.2 Destination pointed by a pointer shall be referenced after checking that the pointer is not the null pointer.

Preference guide	
Rule specification	

Compliant example

```
if (p != nullptr)
  *p = 1;
```

Non-compliant example

```
*p = 1;
```

[Related rules] R3.2.1 R3.3.1

R3.3 Check the interface restrictions when a function is called.

R3.3.1 If a function returns error information, then that error information shall be tested.

Preference guide	
Rule specification	

Compliant example

```
p = malloc(BUFFERSIZE);
if (p == nullptr)
  // Error handling
else
  *p = '\0';
```

Non-compliant example

```
p = malloc(BUFFERSIZE);
*p = '\0';
```

When a function returns a value, the code that does not use that return value may cause an error. If it is not necessary to reference the return value, consider setting a project-specific rule to clearly indicate the unnecessity of referencing, such as, by casting to void.

[Related rules] R3.2.1 R3.2.2 R3.5.1 R3.5.2

R3.3.2 The restrictions of the parameters shall be checked before starting the function processing.

Preference guide	
Rule specification	

Compliant example

```
int func(int para) {
  if (!((MIN <= para) && (para <= MAX)))
    return range_error;
  // Normal processing.
  …
}
```

Non-compliant example

```
int func(int para) {
  // Normal processing.
  …
}
```

Whether to check the restrictions of the parameters at the caller or callee side or at the called side depends on how the interface is designed. Nevertheless, to avoid forgetting to check the parameter restrictions, make it a rule to perform the same check to all the parameters at once at the side to which the function is called.
In case the function to be called cannot be modified, such as, when the function is called from the library, create a wrapper function.

Example of a wrapper function:

```
int func_with_check(int arg) {
  // Return range_error if arg is violating the parameter restrictions.
  // If not, call func and return the result.



// Make a function call by using a wrapper function.
if (func_with_check(para) == range_error) {
  // Error processing.
}
```

R3.4 Do not perform recursive calls.

R3.4.1 Functions shall not call themselves, either directly or indirectly.

Preference guide	
Rule specification	

Compliant example

—

Non-compliant example

```
unsigned int calc(unsigned int n) {
  if (n <= 1) {
    return 1;
  }
  return n * calc(n - 1);
  }
```

Since the stack size used at runtime for recursive calls cannot be predicted, there is a risk of stack overflow.

R3.5 Pay attention to branch conditions and describe how to handle cases that do not follow the predefined conditions when they occur.

R3.5.1

The else clause shall be written at the end of an if-else if statement. If it is known that the else condition does not normally occur, the description of the else clause shall be either one of the following:
《 (i) Unexpected condition handling process shall be written in the else clause.
(ii) A comment specified by the project shall be written in the else clause. 》

Preference guide	○
Rule specification	Define

Compliant example

```
// else clause of an if-else if statement
// in case where the else condition does not
// normally occur.
if (var1 == 0) {
  …
} else if (0 < var1) {
  …
} else {
    // Write an unexpected condition
  …// handling process.
}
…
if (var1 == 0) {
  …
} else if (0 < var1) {
  …
} else {
  // NOT REACHED
}
```

Non-compliant example

```
// if-else if statement without the else
// clause
if (var1 == 0) {
  …
} else if (0 < var1) {
  …
}
```

If there is no else clause in an if-else if statement, it is not clear whether the programmer has forgotten to write the else clause or deliberately left out the else clause because the else condition does not occur. Even if it is known that the else condition does not normally occur, the behavior of the program when an unexpected condition occurs can be specified by writing the else clause as follows:

(i) Write in the else condition, the behavior taken when an unexpected condition occurs. (Predefine the behavior of the program if by any chance the else condition occurs).

(ii) Write a project-specific comment like // NOT REACHED that clearly indicates that the else condition does not occur to express that the else clause was not written because it was forgotten. Such comment will improve the readability of the program.

[Related rules] R3.3.1 R3.5.2

R3.5.2

The `default` clause shall be written at the end of a `switch` statement. If it is known that the `default` condition does not normally occur, the description of the `default` clause shall be either one of the following:
《 (i) Unexpected condition handling process shall be written in the *default* clause.
(ii) A comment specified by the project shall be written in the `default` clause. 》

Preference guide	○
Rule specification	Define

Compliant example

```
// default clause of a switch statement in
// case where the default condition does not
// normally occur.
switch(var1) {
case 0:
  …
  break;
case 1:
  …
  break;
default:
  // Write an unexpected condition
  // handling process.
  break;
}
…
switch(var1) {
case 0:
  …
  break;
case 1:
  …
  break;
default:
  // NOT REACHED
  break;
}
```

Non-compliant example

```
// switch statement without the default
// clause
switch(var1) {
case 0:
  …
  break;
case 1:
  …
  break;
}
```

If there is no `default` clause in a `switch` statement, it is not clear whether the programmer has forgotten to write the `default` clause or deliberately left out the `default` clause because the `default` condition does not occur. Even if it is known that the `default` condition does not normally occur, the behavior of the program when an unexpected condition occurs can be specified by writing the `default` clause as follows:

(i) Write the behavior under unexpected conditions in the `default` condition (Predefine the behavior of the program if by any chance the `default` condition occurs).

(ii) Write a project-specific comment like `// NOT REACHED` that clearly indicates that the `default` condition does not occur to express that the `default` clause was not written because it was forgotten. Such comment will improve the readability of the program.

[Related rules] R3.3.1 R3.5.1 M3.1.4

R3.5.3 Equality operators (== !=) shall not be used for comparison of loop counters.

Preference guide	
Rule specification	

Compliant example

```
void func() {
  int i;
  for (i = 0; i < 9; i += 2) {
    …
```

Non-compliant example

```
void func() {
  int i;
  for (i = 0; i != 9; i += 2) {
    …
```

If the amount of change of the loop counter is not 1, an infinite loop may occur. Therefore, for comparison to determine the number of loop iterations, do not use the equality operators (== !=). (Instead use <= >= < >.)

Note) This rule does not apply to iterators.

```
    for (itr = v.begin(); itr != v.end(); ++itr)
```

R3.6 Pay attention to the order of evaluation.

R3.6.1 Variables whose values are changed shall not be referred to or modified in the same expression.

Preference guide	●
Rule specification	

Compliant example

```
f(x , x);
x++;
  Or
f(x + 1, x);
x++;
```

Non-compliant example

```
f(x, x++);
```

Compilers do not guarantee the order of execution (evaluation) of each actual argument in functions with multiple parameters. The arguments may be executed from the right or from the left. In addition, compilers do not guarantee the order of execution of the left-hand and right-hand sides of binary operations like + operation. Therefore, if the same object is updated and referenced in a sequence of arguments or binary operation expressions, the execution result is not guaranteed. A case where the execution result is not guaranteed is called a side effect problem. Do not write code that causes such side effect problems.
This rule, however, does not prohibit descriptions, such as, those shown below which do not cause the side effect problem.

```
x = x + 1;
x = f(x);
```

[Related rules] R3.6.2 M1.8.1

R3.6.2 Function calls with side effects and `volatile` variables shall not be described more than once in a sequence of actual arguments or binary operation expressions.

Preference guide	○
Rule specification	

Compliant example

```
Compliant example of (1)
extern int G_a;
x = func1();
x += func2();
…
int func1(void) {
  G_a += 10;
  …
}
int func2(void) {
  G_a -= 10;
  …
}

Compliant example of (2)
volatile int v;
y = v;
f(y, v);
```

Non-compliant example

```
Non-compliant example of (1)
extern int G_a;
x = func1() + func2();  // With side effect
                        //problem
…
int func1(void) {
  G_a += 10;
  …
}
int func2(void) {
  G_a -= 10;
  …
}

Non-compliant example of (2)
volatile int v;
f(v, v);
```

Compilers do not guarantee the order of execution (evaluation) of each actual argument in functions with multiple parameters. The arguments may be executed from the right or from the left. In addition, compilers do not guarantee the order of execution of the left-hand and right-hand sides of binary operations like + operation. Therefore, the execution results of two or more function calls with side effects and `volatile` variables in a sequence of arguments or binary operation expressions may not be guaranteed. Such unsafe descriptions must be avoided.

[Related rules] R3.6.1 M1.8.1

R3.6.3 `sizeof` operator shall not be used in expressions that have side effect.

Preference guide	●
Rule specification	

Compliant example

```
x = sizeof(i);
i++;
```

Non-compliant example

```
x = sizeof(i++);
```

The expression in parenthesis of `sizeof` operator is used only for finding the size of the expression type, and is not executed. Therefore, even when ++ operator like `sizeof(i++)` is described, `i` is not incremented.

R3.7 Pay attention to the behavior of classes.

R3.7.1 In the class that manages resources, copy constructor, copy assignment operator and destructor shall be defined.

Preference guide	○
Rule specification	

Compliant example

```
Compliant example of (1)
class CLS {
private:
  int *cls_px;  // Resources managed by CLS
                   class.
public:
  CLS() : cls_px(new int(0)) { }
  ~CLS() {delete cls_px;}
  // Compliant: copy constructor and copy
  // assignment operator are provided.
  CLS(const CLS &cls):
    cls_px(new int (*(cls.cls_px))) { }
  // Memory is allocated.
  CLS &operator = (const CLS &cls) {
    cls_px = new int(*(cls.cls_px));
  // Memory is allocated.
    }
};\

Compliant example of (2)
class CLS {
public:
  CLS() : cls_px(new int(0)) { }
  ~CLS() {delete cls_px;}
private:
  // Compliant: Copy constructor and copy
  // assignment operator are declared
  // private to prevent them being used.
  CLS(const CLS &cls);
  CLS &operator = (const CLS &cls);
  int *cls_px;
};

Compliant example of (3)
class CLS {
public:
  CLS() : cls_px(new int(0)) { }
  ~CLS() {delete cls_px;}
  // Compliant: Copy constructor and copy
  // assignment operator are declared as
  // =delete to prevent them from
  // being generated.
  CLS(const CLS &cls) = delete;
  CLS &operator = (const CLS &cls);
  int *cls_px = delete;
};
```

Non-compliant example

```
class CLS {
private:
  int *cls_px;  // Resources managed by CLS
                // class.
public:
  CLS() : cls_px(new int(0)) { }
  ~CLS() {delete cls_px;}
  // Non-compliant: This class manages
  // resources (cls_px) , but copy constructor
  // and copy assignment operator are not
  // described. As a result, the following
  // default copy constructor and copy
  // assignment operator are generated
  // automatically by the compiler.
  CLS(const CLS &c) : cls_px(c.cls_px){    }
  // address is copied.
  CLS &operator = (const CLS &cls) {
  cls_px = cls.cls_px;
  // address is copied.
  }
};
```

In the class that manages resources, such as, memory and files, it is common that the constructor is defined to allocate resources, destructor to release them, and copy constructor or copy assignment operator to copy them.
Therefore, if any of them are not defined, there is a risk of problems being caused by forgetting to write the code to execute the required resource management. In case of a program that has been written not to execute copying, declaring the copy constructor and copy assignment operator as `private` will prevent the compiler from generating them automatically and enable it to output an error when unintended copy is detected while compiling. (See Compliant example of (2).) According to C++11, the generation of copy constructor and copy assignment operator can be prevented declaring copy constructor and copy assignment operator as `=delete` to specify that the function is deleted. (See Compliant example of (3).)

[Related rule] E1.1.3

R3.7.2 Virtual destructor shall be declared in the base class.

Preference guide	○
Rule specification	

Compliant example

```
class Base {
public:
  virtual ~Base() { };  // Compliant
  virtual void show()
  {cout << "Base" << endl;}
};
class Derived : public Base {
public:
  virtual ~Derived()  { }
  virtual void show () override
  {cout << "Derived" << endl;}
};
Base *bp = new Derived;
bp -> show();
delete bp;  // Derived::~Derived() is called.
```

Non-compliant example

```
class Base {
public:
  ~Base() {};  // Not a virtual destructor.
  virtual void show()
  {cout << "Base" << endl;}
};
class Derived : public Base {
public:
  ~Derived()  {}
  virtual void show() override
  {cout << "Derived" << endl;}
};
Base *bp = new Derived;
bp -> show();
delete bp;  // Undefined behavior.
```

In case the destructor of the base class is not virtual, the behavior to delete the object of the derived class via the pointer to the base class is undefined. No behavioral problem will occur even when the destructor of the base class is not virtual, if the object of the derived class is not deleted via the pointer to the base class. This rule takes account of cases when the code is modified.

Reference Declaring the destructor of the base class as protected when the destructor of the virtual class is not declared will enable the compiler to detect the deletion of objects of the derived class via the pointer to the bass class as error at compile time.

R3.7.3

Copy assignment operator and move assignment operator shall be defined to comply with the following rules that state that:

1. **Copy assignment operator and move assignment operator shall return a self-reference.**
2. **Copy assignment operator shall declare in the form of either "`T&operator=(constT&)`" or "`T&operator=(T)`". Move assignment operator shall declare in the form of "`T&operator=(T &&)`". Their return type shall not be `const`-qualified.**
3. **Copy assignment operator shall be capable of self-assignment.**

Preference guide	○
Rule specification	

Compliant example

```
Compliant example of (1)
class C0 {
public:
  C0 &operator = (const C0 &rhs) {
    …;
    return(*this);}
}

C0 c00, c01, c02;
c00 = c01 = c02;  // Compliant

Compliant example of (2)
class C1 {
public:

  C1 &operator = (const C1 &rhs) { … }
                                // Compliant
};
void func(C1 &ref) { … }

C1 c10, c11;
func(c10 = c11); // Compliant

Compliant example of (3)
class C2 {
private:

  char *data;
public:

  C2(char *str) {data = strdup(str);}
  C2 &operator = (const C2 &rhs) {

    if (&rhs == this) return; // Self-
                              // assignment
                              // is checked
    free(data);
    data = strdup(rhs.data);
  }
};

C2 c20("C++");
c20 = c20;  // Compliant
```

Non-compliant example

```
Non-compliant example of (1)
class C0 {
public:
  void operator = (const C0 &rhs) { … }
                  // Non-compliant: Return
                  // value type is not C0 &.
}

C0 c00, c01, c02;
c00 = c01 = c02;  // Non-compliant: Compile
                  // error.

Non-compliant example of (2)
class C1 {
public:
  const C1 &operator = (const C1 &rhs) { … }
                 // Non-compliant: Return
                 // value type is not C1 &.
};
void func(C1 &ref) { … }

C1 c10, c11;
func(c10 = c11);  // Non-compliant: Compile
                  // error.

Non-compliant example of (3)
class C2 {
private:

  char *data;
public:

  C2(char *str) {data = strdup(str);}
  C2 &operator = (const C2 &rhs) {
                 // Non-compliant: Self-
                 // assignment is not
                 // checked.
    free(data);
    data = strdup(rhs.data);
  }
};

C2 c20("C++");
c20 = c20;       // Non-compliant: Invalid
                 // memory is referenced.
```

1. If the copy assignment operator does not return a self-reference, the behavior of multiple assignments (e.g.: `a = b = c`) will not be the same as the behavior of the assignment operator to intrinsic type (like char and `int`).
2. When the return type of copy assignment operator is *const*-qualified, the behavior of multiple assignments (e.g.: `a = b = c`) will not be the same as the behavior of the assignment operator to intrinsic type (like `char` and `int`). The type resulting from assignment operator's behavior with intrinsic type is *T*, and not `const` *T*. Moreover, *T*-type object will not be able to be stored in the container provided by the standard library (because it does not meet the container requirements).
   In C++11, rvalue reference has been introduced as a new feature to reduce the copy overhead of a temporary object, If reducing the copy overhead of a temporary object is important, move assignment operator may be used to take rvalue reference as an argument.
3. Unexpected error may occur when the copy assignment operator is not assuming that self-assignment may occur. Suppose the memory pointed by a member is deleted within the copy assignment operator. When self-assignment occurs in this situation, deleted memory may be referenced.

【Reference materials for those wanting to know more in detail about this rule】

- Effective Modern C++　　Chapter 5

R3.7.4 Default parameter values shall not be changed when overriding virtual functions.

Preference guide	○
Rule specification	

Compliant example

```
class Base {
private:
  string color;
public:
  virtual void set_color(string c = "black")
    {color = c;}
  string get_color() { return color; }
};
class Derived : public Base {
public:
  virtual void set_color(string c = "black")
  override{
  // Compliant
    Base::set_color(c);
  }
};
Derived d;
Base *bp = &d;     // Static type is Base *.
bp->set_color();   // Derived::
                   // set_color("black")
                   // is called.
cout << bp->get_color() << endl;
                   // "black" is displayed.
Derived *dp = &d;  // Static type is Derived *.
dp->set_color();   // Derived::
                   // set_color("black")
                   // is called.
cout << dp->get_color() << endl;
                   // "black" is displayed.
```

Non-compliant example

```
class Base {
private:
  string color;
public:
  virtual void set_color(string c = "black")
    {color = c;}
  string get_color() {return color;}
};
class Derived : public Base {
public:
  virtual void set_color(string c = "white")
  override{
  // Non-compliant: Default parameter value
  // is changed.
    Base::set_color(c);
  }
};
Derived d;
Base *bp = &d;     // Static type is Base *.
bp->set_color();   // Derived::
                   // set_color("black")
                   // is called.
cout << bp->get_color() << endl;
                   // "black" is displayed.
Derived *dp = &d;  // Static type is Derived *.
dp->set_color();   // Derived::
                   // set_color("white")
                   // is called.
cout << dp->get_color() << endl;
                   // "white" is displayed.
```

The default parameter value of a virtual function is not the value of the default parameter of the function that is actually called, but is the value of the default parameter of the function that is determined by the static type (see the above examples). That is why the value of the default parameter of the virtual function that is called differs from the value of the default parameter of the called function.

R3.7.5 Non-virtual function shall not be redefined in the derived class.

Preference guide	●
Rule specification	

Compliant example

```
class Base {
public:
  virtual void disp() { … }
  …
};
class Derived : public Base {
public:
  virtual void disp() override { … }
                              // Compliant
};

Derived d;
Base *bp = &d;
bp->disp();  // Derived::disp() is called.
```

Non-compliant example

```
class Base {
public:
  void disp() { … }
  …
};
class Derived : public Base {
public:
  void disp() { … }  // Non-compliant
};

Derived d;
Base *bp = &d;
bp->disp();  // Base::disp() is called.
```

Even when the non-virtual function is redefined in the derived class, the behavior will not become polymorphic.
`bp->disp()` written in the non-compliant example shown above calls `disp()` of the base class even when a pointer to an object of the derived class is set as pointer `bp`, because `disp()` is not a virtual function.

Reference Difference between pure virtual function, virtual function and non-virtual function Pure virtual function: Declares the interface. (Always defined in the derived class.)
Virtual function: Defines the default behavior. (Redefined when the default behavior is changed in the derived class.)
Non-virtual function: Defines the behavior that remains the same between the base class and derived class. (Not redefined in the derived class.)

[Related rule] R3.7.7

R3.7.6 Pass by reference or pass by pointer shall be used to set an object for polymorphic behavior as the function argument.

Preference guide	●
Rule specification	

Compliant example

```
class Base {
public:
  virtual void show(void) const;
  …
}
class Derived : public Base {
public:
  virtual void show(void) const override;
  …
};
void calc(const Base &bar) {  // Compliant:
                              // Pass by
                              // reference.
  bar.show();  // show() to be called is
               // determined by the dynamic
               // type of bar.
}
```

Non-compliant example

```
class Base {
public:
  virtual void show(void) const;
  …
}
class Derived : public Base {
public:
  virtual void show(void) const override;
  …
};
void calc(Base bar) {  // Non-compliant:
                       // Pass by value.
  bar.show();  // Base::show() is always
               // called.
}
```

When a derived class object is received as the function argument of base class type through pass by value, object slicing occurs. This is a problem where the additional attributes of the derived class object are sliced or ignored. Due to this problem, the received function cannot make the object behave polymorphically.

[Related rule] R3.8.7

R3.7.7 Override keyword shall be written for overriding of virtual function to occur.

Preference guide	○
Rule specification	

Compliant example

```
class Base {
public:
  virtual void vfunc1(float);
  virtual void vfunc2(void) const;
  …
};
class Derived : public Base {
public:
  virtual void vfunc1(int) override;
       // Compliant: The programmer will
       // be able to know for sure that
       // overriding did not occur because
       // the compiler will output an error.
  virtual void vfunc2(void) override;
       // Compliant: The programmer will be
       // able to know for sure that
       // overriding did not occur because
       // the compiler will output an error.
  …
};
```

Non-compliant example

```
class Base {
public:
  virtual void vfunc1(float);
  virtual void vfunc2(void) const;
  …
};
class Derived : public Base {
public:
  virtual void vfunc1(int);
        // Non-compliant: The compiler may
        // not always output an error.
  virtual void vfunc2(void);
        // Non-compliant: The compiler may
        // not always output an error.
  …
};
```

In the non-compliant example, `vfunc1` function was written, expecting that overriding would occur. But it will not occur due to the difference in argument type. Since the compiler may not always output a warning, there is a risk of unintended behavior when the compiler leaves the error unnoticed.
When there is a need for overriding, write the `override` keyword as shown in the compliant example to make sure that the compiler will output an error when overriding does not occur.

Below is an example where the programmer forgot to write the `virtual` keyword to the member function in the base class. Even in this case, writing the `override` keyword to the member function in the inheritance class inherited class will ensure that the compiler will output an error and enable the programmer to notice that the `virtual` keyword was not written.

```
class Base {
  void vfunc(void); //virtual is missing from the description.
                    //The correct description should be
                    //virtual void vfunc(void).
};
class Derived : public Base {
  void vfunc(void) override; //override keyboard is written.
};
```

【Reference materials for those wanting to know more in detail about this rule】

- Effective Modern C++　　Item 12

[Related rule] R3.7.5

R3.8 Pay attention to the behavior of exceptions.

R3.8.1

(1) Exception handling shall not be used.

(2) When using exception handling,《its method of use in the project shall be specifically defined》.

Preference guide	
Rule specification	Choose / Define

Compliant example

```
Compliant example of (1)
int *p = new(nothrow) int(10);
             // Compliant: Even when failing
             // in new, exception is not
             // thrown and NULL is returned.
if (p == NULL) {
  return ERROR;  // Compliant: Handled without
                 // throwing exception.
}
…
```

Non-compliant example

```
Non-compliant example of (1)
try {
  int *p = new int(10);
                // Non-compliant: When
                // failing in new, exception
                // is thrown.
  …
}
catch(bad_alloc) {
  …
}
```

Exception handling is normally used for error notification, and especially for notifying the error between functions. By using exception handling, the code for error handling can be separated. As a result, the readability of the written code will improve. The problem with exception handling in general is that its behavioral complexity may have an adverse effect on efficiency, among others. Therefore, a rule specifying when to allow the use of exception handling should be defined in each project.

Example rule:
Exception handling shall be used only for error notification. Moreover, the list of exceptions that are allowed to be handled shall be created, and any exceptions other than those listed shall not be handled.

【Reference materials for those wanting to know more in detail about this rule】

- C++ Coding Standard Item 72
- Effective C++ Item 29
- More Effective C++ Item 15

[Related rules] R3.8.2 R3.8.3 R3.8.4 R3.8.5 R3.8.6 R3.8.7 R3.8.8 R3.8.9

R3.8.2 Exception specification shall not be described.

Preference guide	
Rule specification	

Compliant example

```
void func(int x,int y){
       // Exception specification is not
       // escribed. There is no limitation to
       // the exception that is thrown.
  if  (…) {throw  1；}
  …
  if  (…) {throw  "ERROR"；}
  …
}

void  func2(){
  try {
    func1();
  }
  catch(...){
    // Since exception specification is not
    // described in func1(), any exception
    // thrown by func() can be caught here,
    // regardless of whether the exception
    // thrown is an int or char* type. This
    // behavior is easy to understand and
    // also to deal with.
```

Non-compliant example

```
void func1() throw(int){
       // Exception specification isdescribed.
       // Exceptions other than the int type
       // will not be thrown.
  if (…){throw  1；}
  …
  if (…){
    throw "ERROR"；
       // "ERROR" is not an int type but will
       // not be detected as a compile error.
       // As a result, "ERROR" will not be
       // thrown, unexpected() will be called,
       // and the program will terminate by
       // default. Such behavior is misleading
       // and difficult to deal with.
  }
  …
}

void func2(){
  try{
    func1();
  }
  catch(...){
       // Since char* type is not described
       // as exception specification in
       // function func1(), unexpected() will
       // be called when char* type "ERROR"
       // is thrown by func1. As a result,
       // this exception cannot be caught
       // here.
```

Under C++ language specification, compile error will not occur even when a function throws an exception that is not described in the exception specification. What will happen is that `std::unexpected` will be called, and `catch(...)` in the function that called the exception throwing function will not be able to catch the thrown exception.
The default of `unexpected()` is `std::terminate`, and normally, `abort()` will be called.
This behavior is misleading to programmers, and makes it difficult for them to deal with it correctly.
C++11 does not recommend the use of the keyword throw for exception specification. Instead, it allows the use of the keyword `noexcept` to specify that the function will not throw exceptions. However, since this check is also made at the time of execution, the same problem described in the preceding paragraph above must be taken into consideration.

【Reference materials for those wanting to know more in detail about this rule】
- C++ Coding Standard　　Item 45

[Related rule] R3.8.1

R3.8.3 NULL shall not be thrown.

Preference guide	●
Rule specification	

Compliant example

```
try {
  throw 0;  // Compliant
  …
}
catch(char *e) {
  // 0 is not caught here.
}
catch(int e) {
  // 0 is caught here.
}
```

Non-compliant example

```
try {
  throw NULL;  // Non-compliant
  …
}
catch(char *e) {
  // NULL is not caught here.
}
catch(int e) {
  // NULL is caught here.
}
```

`throw NULL` is the same as `throw 0` and is not caught by the pointer type handler.
The purpose of this rule is to prevent writing a description to `catch NULL` with a pointer type handler.
In C++11, `nullptr` is introduced as null pointer constant. The type of `nullptr` is `nullptr_t` which differs with any pointer type, and cannot be captured by pointer type handler.

[Related rules] R3.8.1　R3.8.4

R3.8.4 Pointer shall not be thrown as an exception.

Preference guide	●
Rule specification	

Compliant example

```
void func() {
  C obj;
  throw obj;     // Compliant: Throw by value.
}

try {
func();
}
catch(C &ref) {  // Catch by reference.
  …
}
```

Non-compliant example

```
void func1() {
  …
  throw new C();  // Non-compliant: Throw by
                  // pointer.
}

try {
  func1();
}
catch(C *ptr) {
  …  // Cannot determine whether there is a
     // need to delete the object or not.
}

void func2() {
  C obj;
  throw &obj;     // Non-compliant: Throw by
                  // pointer.
}

try {
  func2();
}
catch(C *ptr) {
  …               // Vanished object is
                  // referenced.
}
```

When a pointer to an object is thrown, the block that catches it has to determine whether there is a need to delete the object or not, and the operation to delete the object may easily be forgotten. Moreover, when a pointer to an object on the stack is thrown as an exception, the block that catches it may reference the object that has already vanished.

[Related rules] R3.8.1 R3.8.3 R3.8.7

R3.8.5 Destructor shall not throw exceptions.

Preference guide	●
Rule specification	

Compliant example

```
C::~C() {     // Compliant: Exception is not
              // thrown.
  try {
    func1();  // Function that may throw
              // exceptions
  }
  catch(...) {
              // Code that deals with an
              // exception.
  }
}

void func2(){
  C obj; …
}             // Destructor (C::~C) is called.
```

Non-compliant example

```
C::~C() {    // Non-compliant: Exceptions may
             // be thrown.
  func1();   // Function that may throw
             // exceptions
}

void func2(){
  C obj;
  …
}            // Destructor (C::~C) is called.
```

There are cases when an exception is thrown from the destructor, such as:

- When a `throw` expression is used to throw the exception
- When a function that has a possibility of throwing exceptions is called.

The function that is called when an exception is thrown from a destructor may be `terminate` function. The behavior when the program is forced to terminate by `terminate` function will be either calling `abort()` function by default or calling the process specified by `set_terminate` function. Therefore, any exception thrown by the destructor must all be caught in the destructor. In other words, no exception shall be thrown out of the destructor.

[Related rule] R3.8.1

R3.8.6 No argument shall be written in the *throw* expression when rethrowing an exception.

Preference guide	●
Rule specification	

適合例

```
void func() {
  try {
    …
  }
  catch(MyError) {
    …
    throw;
  }
}
```

Non-compliant example

```
void func() {
  try {
    …
}
  catch(MyError) {
    …
    throw MyError;
  }
}
```

When an exception thrown from the derived class is caught by its base class, and if the exception object of that base class is written in a `throw` expression when that exception is rethrown, the exception object of the base class will be thrown and polymorphic behavior will be lost.
Therefore, to rethrow an object of a derived class, do not write the argument in the `throw` expression.

[Related rule] R3.8.1

R3.8.7 Exception object shall be caught by reference.

Preference guide	●
Rule specification	

Compliant example

```
class BaseError {
  …
  virtual void vfunc() { … }
};

class DerivedError : public BaseError {
  …
  virtual void vfunc() override { … }
};

void func1(){
  DerivedError obj;
  throw obj; // DerivedError is thrown.
}
  try {
    func1();
  }
  catch(BaseError &ref) {
    ref.vfunc(); // Compliant:
                 // DerivedError::func is
                 // called.
  }
```

Non-compliant example

```
class BaseError {
  …
  virtual void vfunc() { … }
};

class DerivedError : public BaseError {
  …
  virtual void vfunc() { … }
};

void func1(){
  DerivedError obj;
  throw obj; // DerivedError is thrown.
}
  try {
    func1();
  }
  catch(BaseError e) {
    e.vfunc(); // Non-compliant:
               // BaseError::func is called.
  }
```

When an exception is caught by value, the exception object is copied and converted to an object of its base class. As a result, the behavior of the object may change. When an exception is caught by pointer, there is a need to confirm that the memory pointed by the pointer is freed after completing the exception handling. To prevent these problems, throw the exception by value and catch it by reference.
The pass and receive of an exception object in exception handling is similar to pass and receive of a parameter in a function call, but in exception handling, the object is copied even when it is caught by reference (whereas, in function call, the object is not copied when it is passed by reference).Therefore, for example, even when an object in stack memory is caught by reference, it will not cause such problems as referencing invalid memory.

[Related rules] R3.7.6 R3.8.1 R3.8.4

R3.8.8 Exception handlers shall be written in the order of derived class, base class and "..." (that catches all the exceptions).

Preference guide	●
Rule specification	

Compliant example

```
class BaseError { … };
class DerivedError : public BaseError { … };

void func() {
  try {
    …
  }
  catch(DerivedError) {
       // Compliant: In the order of derived
       // class → base class
    …
  }
  catch(BaseError) {
    …
  }
  catch(Exception) {
    …
  }
  catch(...) {          // Compliant: "..."
                        // is the end.
    …
  }
}
```

Non-compliant example

```
class BaseError { … };
class DerivedError : public BaseError { … };

void func() {
  try {
    …
  }
  catch(BaseError) {
       // Non-compliant: In the order of
       // base class → derived class
    …
  }
  catch(DerivedError) {
     … // DerivedError is the derived
       // class of BaseError, so this
       // handler is not executed.
  }
  catch(...) {      // Non-compliant: "..."
                    // is not the end.
    …
  }
  catch(Exception) {
  …    // "catch(...)" catches all the
       // exception, so this handler is not
       // executed.
  }
}
```

Handlers for catching exceptions are checked in the order they are written. For example, when an object of a derived class is thrown, it will be caught by the handler of the base class, if this handler is written before the handler of the derived class. Moreover, if "...", which catches all the exceptions, is used as a handler, be sure to write "..." as the last handler.

[Related rule] R3.8.1

R3.8.9 All the exceptions shall be caught without any omission in the main function and thread start function.

Preference guide	●
Rule specification	

Compliant example

```
int func1();  // Function that throws
              // exceptions
int func2() {
  try {
    return func1();
  }
  catch(...) {  // Compliant:All The
                // exceptions are caught.
    …
    return 0;
  }
}
void func3() {
  …
  int x = func2();
  …
}

int y = func2();  // Compliant: Exceptions
                  // are not thrown.

int main() {
  try {
    …
    std::thread th(func3);
       // Compliant: Exceptions are not
       // thrown.
    …
  }
  catch(Error) {
    …
  }
  catch(...) {  // Compliant: All the
                // exceptions are caught.
    …
  }
}
```

Non-compliant example

```
int func1();  // Function that throws
              // exceptions
void func3() {
  try {
    …
    int x = func1();
    …
  }
  catch(...) {  // All the exceptions are
                // caught.
    …
  }
}

int y = func1();  // Non-compliant:
                  // Exceptions are not
                  // caught.

int main() {
  try {
    …
    std::thread th(func3);
                // Non-compliant: Not all
                // exceptions are caught.
    …
  }
  catch(Error) { // Non-compliant: Not all
                 // exceptions are caught.
    …
  }
}
```

When any exception catch clause is missing in the main function, terminate function will be called. As explained earlier in R3.8.5, the behavior when the program is forced to terminate by terminate function will be either calling `abort()` function by default or calling the process specified by `set_terminate` function. Therefore, the backward compatibility can be secured by writing `catch(...)` in the main function so that the program can be properly terminated.
Since exceptions thrown when initializing the global variables within the definition cannot be caught, there is a need to be careful not to create any exceptions when global variables are initialized.

[Related rule] R3.8.1

R3.9 Pay attention to the behavior of templates.

R3.9.1 In case the template formal parameter is referenced by pointer, template specialization shall be prepared.

Preference guide	●
Rule specification	

Compliant example

```
template <class T> class TCLS {
public:
  bool tfunc(T t1, T t2) {
    return t1 == t2;
  }
};
template <class T> class TCLS<T*> {
public:
  bool tfunc(T *t1, T *t2) {
    return *t1 == *t2;
  }
};

void f(int *px, int *py) {
  TCLS<int*> tc;
  tc.tfunc(px, py);
```

Non-compliant example

```
template <class T> class TCLS {
public:
  bool tfunc(T t1, T t2) {
    return t1 == t2;
  }
};

// Specialization for the pointer is not
// prepared.
void f(int *px, int *py) {
  TCLS<int*> tc;
  tc.tfunc(px, py);
       // Non-compliant: The addresses are
       // compared instead of comparing the
       // value as intended
```

When a template whose formal parameter is not considered to be referenced by pointer is used as a pointer by the template argument, it will not be determined as a compile error and may cause an unintended behavior.

[Related rule] M1.2.6

R3.10 Pay attention to the behavior of lambda expressions.

R3.10.1 In lambda expressions, default capture mode shall not be used, and all the local names used shall be written explicitly.

Preference guide	
Rule specification	

Compliant example

```
void f1() {
  …
  auto func = [ ] (int x, int y) {return x > y; };
    // Compliant: Local names are not used.
  …
}

void f2() {
  int y = ...;
  …
  auto func = [y] (int x) { return x > y; } ;
    // Compliant: Local name (auto variable)
    // is used and is explicit.
  …
}
```

Non-compliant example

```
void f() {
  int y = ...;
  …
  auto func = [=] (int x) { return x > y; } ;
    // Non-compliant: Local name (auto
    // variable) is used but is not explicit.
  …
}
```

Lambda expression introduced in C++11 helps define functions easily. Unlike ordinary functions, lambda expression makes it possible to write within the function. By specifying the lambda capture ([ ] in the beginning of the lambda expression), the local names of the function (automatic variable, parameter, etc.) can be accessed. However, lambda expressions need to be used carefully. For example, if the lambda expression is going to exist longer than the function (in which it is written), such as, when it is used after saving it in a global variable, the space for a captured automatic variable is gone by the thime it is accessed as a referance, resulting in access violation. (The behavior would be the same as the bug that is used outside the function after the function returns the reference and pointer to the automatic variable.)

```
void addDivisorFilter() {
  ...
auto divisor = ...
filters.emplace_back([&](int value) { return value % devisor == 0; } );
      // The function defined by lambda expression is registered in
      // filters, but it uses the divisor of auto variable as the reference.
      // This space extinguishes after passing addDivisorFilter
      // function. The filtering process using filters is executed outside
      // addDivisorFilter function. As a result, the space that has
      // already extinguished is referenced.
}
```

This is a rule concerning the method of lambda capture specification. Among the three ways of specifying the lambda capture listed below as 1), 2) and 3), this rule is aimed at minimizing coding errors by encouraging the users to use 3) and write out all the names used so that they can be easily checked at the time of review, instead of using 1) or 2), which are both default capture modes (access by reference or access by value, respectively).

1) `[&]`: It makes all the local names available as “reference”.
2) `[=]`: It make all the local names available as “value”.
3) [*capture_list*]: It makes only the local names written in this name list available. If `&` is placed immediately before the name, that name will be available as “reference”. If nothing is placed before the name, that name will be available as “value”.

In 1), the variable is accessed as reference by default. Therefore, unintended reference access may occur as mentioned above. In 2), the variable is accessed as value by default, but in the case with member function, *this* will be accessible by default. What this means is that reference access to the data members of a class will be possible through *this*, and unintended reference access may occur. Those interested in learning more about this problem should read the explanation provided in Item 31 of Effective Modern C++.

Since the specifications of lambda expressions are difficult to understand, lamba should be used only after gaining a good knowledge about its specifications through reliable reference materials, such as, the ones listed below.

【Reference materials for those wanting to know more in detail about this rule】

- Effective Modern C++ Item 31 Item 32 Item 33 Item 34
- Google C++ style guide Lambda expressions

R3.11 Be careful with how to access the shared data in programs that use threads or signals.

R3.11.1 `std::atomic` shall be used for concurrent processing instead of volatile.

Preference guide	
Rule specification	

Compliant example

```
std::atomic<int> v(0);   // Compliant
…
v++; // Processed indivisibly.
…
```

Non-compliant example

```
volatile int v = 0;      // Non-compliant
…
v++; // Not processed indivisibly.
…
```

For concurrent processing or asynchronous signal processing, there is a need to properly reflect the result of updated data to other threads. The memory model assumed in C++11 does not guarantee the indivisibility and visibility of data refreshed by other threads. Using `volatile` to guarantee them is a mistake. `volatile` is used for preventing the compiler from optimizing the data and does not guarantee indivisibility, etc., in concurrent processing. C++11 support the use of `std::atomic` to guarantee the indivisibility of a single data. It also supports the use of mutex, etc., when there is a need to process more complex data indivisibly.

R3.11.2 The bit fields that may be allocated in the same memory space shall not be accessed by multiple threads or shall be exclusively controlled properly.

Preference guide	
Rule specification	

Compliant example

```
(1)
struct {
  unsigned int flag0 : 1;
  :0,
  unsigned int flag1 : 1;
} s;  // Compliant: flag0 and flag1 are in two
      // different memory spaces.
void fun0() {
  s.flag0 = 1;
}
void fun1() {
  s.flag1 = 1;
}
…
  // fun0 and fun1 are executed by different
  // threads.
  std::thread t0(fun0);
  std::thread t1(fun1);
…

(2)
struct {
  unsigned int flag0 : 1;
  unsigned int flag1 : 1;
} s;  // flag0 and flag1 are in the same
      // memory space.
std::mutex lock;  // mutex is used for
                  // exclusive control
// Compliant: Exclusively controlled
// properly.
void fun0() {
    std::unique_lock<std::mutex> ul(lock);
       // mutex is locked (and unlocked when
       // the function ends).
    s.flag0 = 1;
}
void fun1() {
    std::unique_lock<std::mutex> ul(lock);
       // mutex is locked (and unlocked when
       // the function ends).
    s.flag1 = 1;
}
…
  // fun0 and fun1 are executed by different
  // threads.
  std::thread t0(fun0);
  std::thread t1(fun1);
…
```

Non-compliant example

```
struct {
  unsigned int flag0 : 1;
  unsigned int flag1 : 1;
} s;  // Non-compliant:  flag0 and flag1 are
      // in the same memory space, and are
      // not exclusively controlled
      // properly.
void fun0() {
  s.flag0 = 1;
}
void fun1() {
  s.flag1 = 1;
}
…
  // fun0 and fun1 are executed by different
  // threads.
  std::thread t0(fun0);
  std::thread t1(fun1);
…
```

C++11 defines 1 byte as the smallest memory space to access data. Therefore, if multiple threads access bit fields allocated in the same memory space, the result of data referenced or refreshed in the adjacent bit fields may become incorrect. To avoid this problem, bit field of length zero (0) should be used to allocate the data separately in different memory spaces or the access to bit fields shall be exclusively controlled properly.

【Reference materials for those wanting to know more in detail about this rule】

- CERT C　　CON32-C

[Related rule] P1.3.3

Maintainability

Many embedded software developments require maintenance tasks, including the modification of the software that has already been developed.

There are various reasons for maintenance. For example, maintenance becomes necessary:

- When a bug is found in one part of the released software and must be modified;
- When a new function is added to existing software in response to the market demand toward the product.

When any kind of additional work is carried out on the already developed software as in the above examples, it is important to perform such work as accurately and efficiently as possible to maintain the quality of the software.

This is called “maintainability” in the field of system development. This section clarifies the practices to keep and improve the maintainability of embedded software source code.

- **Maintainability M1 : Keep in mind that others will read the program.**
- **Maintainability M2 : Write in a style that can prevent modification errors.**
- **Maintainability M3 : Write programs simply.**
- **Maintainability M4 : Write in a unified style.**
- **Maintainability M5 : Write in a style that makes testing easy.**

Maintainability M1

Keep in mind that others will read the program.

It is easily conceivable that source code is reused and maintained by engineers who are not the original creators. Therefore, it is necessary to write source code that is easy to understand by taking account of others who will read it later.

Maintainability M1.1	Do not leave unused descriptions.
Maintainability M1.2	Do not write confusingly.
Maintainability M1.3	Do not write in an unconventional style.
Maintainability M1.4	Write in a style that clearly specifies the order of evaluation of operations.
Maintainability M1.5	Explicitly describe the operations that are likely to cause misunderstanding when they are omitted.
Maintainability M1.6	Use one area for one purpose.
Maintainability M1.7	Do not reuse names.
Maintainability M1.8	Do not use language specifications that are likely to cause misunderstanding.
Maintainability M1.9	When writing in an unconventional style, explicitly state its intention.
Maintainability M1.10	Do not embed magic numbers.
Maintainability M1.11	Explicitly state the area attributes.
Maintainability M1.12	Correctly describe the statements even if they are not compiled.

M1.1 Do not leave unused descriptions.

M1.1.1 Unused functions, variables, parameters, `typedefs`, tags, labels or macros shall not be declared (defined).

Preference guide	○
Rule specification	

Compliant example

```
void func(void) {
  …
}
When necessary in case of callback function
int cbfunc1(int arg1, int arg2);
int cbfunc2(int arg1, int);
  // There is no need of using the second
  // parameter (arg2) if the signature of
  // callback function is fixed to int(*)
(int,int).

When necessary in case of virtual function
class Base {
public:
  virtual void func(int arg1, int arg2) = 0:
  …
};
class Derived1 : public Base {
public:
  virtual void func(int arg1, int arg2)
                                    override {
    // The second parameter (arg2) is used.
  }
  …
};
class Derived2 : public Base {
public:
  virtual void func(int arg1, int) override {
    // The second parameter (arg2) is
    // necessary to be consistent with the
    // signature.
  }
  …
};
```

Non-compliant example

```
void func(int arg) {
              //   arg is not used.
  ...
}
```

Declaration (definition) of unused functions, variables, parameters or labels impairs maintainability because it makes it difficult to determine whether the programmer has forgotten to delete them or has made a description error.

However, declaration (definition) of unused parameters in a function is necessary to be consistent with the function signature in case of:

- Callback function;
- Virtual function.

In such cases, make it clear that these parameters are unused by not writing their names.

Reference material for those seeking for more details

- MISRA C++ R0-1-11, R0-1-12

[Related rules] M1.9.1 M4.5.1 M4.7.2

M1.1.2 (1) Sections of code should not be "commented out". (2) For commenting out sections of code, 《the coding rule shall be specified.》

Preference guide	○
Rule specification	Choose Define

Compliant example

```
Compliant example of (2)
  …
// i = i * i;
  j = j * I;
  …
```

Non-compliant example

```
Non-compliant example of (1)

  …
// i = i * i;
  j = j * I;
  …
```

Normally, invalidated sections of the code should not be left in the code as it may impair the code readability.
However, if there is a need to invalidate certain sections of the code by commenting them out, set a rule, for example, to use only // comment for commenting out. Any section of the code can also be invalidated without using comment out by specifying that section in between `#if 0` and `endif#`.

[Related rules] M1.12.1 M4.7.2

M1.2 Do not write confusingly.

M1.2.1

(1) Only one variable shall be declared in one declaration statement (avoid multiple declarations.)
(2) Automatic variables of the same type used for the similar purposes may be declared in one declaration statement, but variables with initialization and variables without initialization shall not be mixed.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1)
int i;
int j;

Compliant example of (2)
int i, j;
int k = 0;

int *p;
int i;
```

Non-compliant example

```
Non-compliant example of (1)
int i, j;

Non-compliant example of (2)
int i, j, k = 0;  // Non-compliant:
                  // A variable with
                  // initialization and
                  // variables without
                  // initialization are mixed
int *p, i;        // Non-compliant:Variables
                  // of different types are
                  // mixed
```

If the declaration is `int *p;`, the type of p is `int*`. However, if the declaration is `int *p, q;`, the type of `q` becomes `int` instead of `int*`.

[Related rule] M1.6.1

M1.2.2

Suffixes shall be added to constant descriptions that can use them to indicate appropriate types. Only an uppercase letter "L" shall be used for a suffix indicating a long type integer constant.

Preference guide	
Rule specification	

Compliant example

```
void func(long int);
…
float f;
long int l;
unsigned int ui;

f = f + 1.0F;  // Explicitly state that it is
               // a float operation
func(1L);      // Description of L should be
               // an uppercase letter
if (ui < 0x8000U) {  // Explicitly state
                     // that it is an
                     // unsigned comparison.
```

Non-compliant example

```
void func(long int);
...
float f;
long int l;
unsigned int ui;
f = f + 1.0;
func(1l); // 1l (numeral "1" and alphabet
          // letter "l") can get easily
          // confused with 11 (numeral "1"
          // and numeral "1" ).
if (ui < 0x8000) {
   ...
```

Maintainability

M1

Basically, when there is no suffix, an integer constant will be an `int` type and a floating constant will be a `double` type. However, when an integer constant value that cannot be expressed with an `int` type is described, its type will be the one that can express that value. Therefore, `0x8000` will be `unsigned int` if `int` is 16 bits, and `signed int` if `int` is 32 bits.
To use it as `unsigned`, it is necessary to explicitly describe "`U`" as the suffix. In addition, in case of a target system where the operation speed differs between floating point number of `float` type and that of `double` type, it should be noted that the operation will be a `double` type when performing operations between a `float` type variable and a floating constant without a suffix "`F`".
For floating constants, writing at least one digit on both sides of the decimal point will make them easily recognizable as floating constants.

[Related rule] M1.8.5

M1.2.3 When expressing a long string literal, successive string literals shall be concatenated without using newlines within the string literal.

Preference guide	
Rule specification	

Compliant example

```
char abc[] = "aaaaaaaa\n"
             "bbbbbbbb\n"
             "ccccccc\n";
```

Non-compliant example

```
char abc[] = "aaaaaaaa\n\
              bbbbbbbb\n\
              ccccccc\n";
```

In C++11, a new feature called raw string literal has been introduced. Raw string literal can be used in markup and regular expressions, and make it possible to write literal expressions without using escape characters. If raw string literal is used to write the non-compliant example of this rule, the code will look like the following:

```
char abc[] = R"(aaaaaaaa
bbbbbbbb
cccccccc)";
```

M1.2.4 A rule specifying how to use the namespace shall be defined.

Preference guide	
Rule specification	Choose

Define a rule on how to reference the name in the namespace by taking account of readability and ease of coding. For example, consider establishing the following usage as the project-specific rule.

- The name in the namespace shall be referenced directly by using declaration or scope resolution operator (::), instead of using the using directive. See below for the compliant and non-compliant examples of this rule:

```
using NS1 :: x ;     // Compliant: using   declaration
using NS1 ;          // Non-compliant : using   directive
```

- Up to five names in the namespace shall be referenced directly by using declaration or scope resolution operator (::), and six or more names in the namespace shall be referenced by using the using directive.

M1.2.5 Namespace definition shall not be nested.

Preference guide	
Rule specification	

Compliant example

```
namespace NS1 {
  int x;
}
namespace NS2 { // Compliant:Namespace that
                   is not nested.
  int y;
}
```

Non-compliant example

```
namespace NS1 {
  int x;
  namespace NS2 { // Non-compliant:Namespace
                     that is nested.
    int x;
    int y;
  }
}
```

Nested namespace definition deep down in the hierarchy impairs readability. Therefore, do not nest namespace definition.
"Namespaces will not be nested more than two levels deep." is the rule defined in JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM [19] and this can be adopted.

M1.2.6 Function template shall not be explicitly specialized.

Preference guide	
Rule specification	

Compliant example

```
template <typename T>
  void func(T) { … }    // #1
template <typename T>
  void func(T *) { … }  // #2

void func(int *) { … }  // Compliant:#3: Non-
                        // template function


int main() {
  int x;
  func(&x);  // #3 is called. (Normal function
             // is selected as a priority
             // over template.)
}
```

Non-compliant example

```
template <typename T>
  void func(T) { … }    // #1
template <typename T>
  void func(T *) { … }  // #2

template <> void func(int *) { … }
             // Non-compliant:#3: Function
             // template is  explicitly
             // specialized. (Specialization
             // of #1 [T = int *]).

int main() {
  int x;
  func(&x);  // #2 is called rather than
             // #3 (because #2 is more
             // restricted than #1).
}
```

By explicitly specializing a function template, a function definition that applies specifically for a particular group of template parameters can be provided. However, unintended function may be called, since overload resolution takes place after selecting the specialized function.

[Related rule] R3.9.1

M1.2.7 If the declaration of a constructor becomes the same as the declaration of a default copy constructor when all the parameters specified with a default value are excluded, such constructor shall not be defined.

Preference guide	●
Rule specification	

Compliant example

```
class C {
public:
  C() { … }
  C(C &x, int i) {  // Compliant:Not the
                    // same declaration as
                    // copy constructor.

    …
  }
  …
};

void func(void) {
  C x;
  C y = x;    // Copy constructor is called.
  C z(y, 1);  // "C::C(C &, int)" is called.
  …
}
```

Non-compliant example

```
class C {
public:
  C() { … }
  C(C &x, int i = 0) {  // Non-compliant:Same
                        // declaration as
                        // copy constructor.

    …
  }
  …
};

void func(void) {
  C x;
  C y = x;  // "C::C(C &, int)" is called.
  …
}
```

If the declaration of a constructor becomes identical to the declaration of a default copy constructor after excluding all of its parameters specified with a default value, that constructor will be handled as a default copy constructor. Unless this point is understood clearly, there is a risk of defining a constructor that may result in unintended calls.

【Reference material for those wanting to know more in detail about this rule】

- JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM Item 77

M1.3 Do not write in an unconventional style.

M1.3.1 Expressions evaluating to true or false shall not be described in `switch (`*expression*`)`.

Preference guide	●
Rule specification	

Compliant example

—

Non-compliant example

```
switch (i_var1 == 0) {
case 0:
  i_var2 = 1;
  break;
default:
  i_var2 = 0;
  break;
}
```

When an expression evaluating to true or false is used in a `switch` statement, the number of branch directions will be two, and the necessity of using the switch statement as a multiway branch command becomes low. Compared to if statements, `switch` statements have a higher possibility of errors, such as, writing the default clause wrongly or missing `break` statements. Therefore, if statements shall be used when there will be two branch directions. The non-compliant example is written with `if` statement as follows:

```
if (i_var1 == 0) {
  i_var2 = 0;
} else {
  i_var2 = 1;
}
```

M1.3.2 The case labels and default label in a `switch` statement shall be described only in the compound statement (excluding nested compound statements) within the body of the `switch` statement.

Preference guide	
Rule specification	

Compliant example

```
switch (x) {
case 1:
  {
   …
  }
  …
  break;
case 2:
  …
  break;
default:
  …
  break;
}
```

Non-compliant example

```
switch (x) { // Compound statement in switch
statement body
case 1 :
  {       // Nested compound statement
case 2 :  // Do not describe case label in
          // nested compound statement.
    …
  }
  …
  break;
default:
  …
  break;
}
```

*M1.3.3 is deleted in C++ Language edition and is vacant. (see the table in Appendix)

M1.4 Write in a style that clearly specifies the order of evaluation of operations.

M1.4.1 Expressions described at the right hand and left hand of && and || operations shall be either expressions that do not include binary operation or expressions enclosed with ( ). However, if only && operations or only || operations are successively combined, it is not necessary to enclose each && and || expression with ( ).

Preference guide	
Rule specification	

Compliant example

```
if ((x > 0) && (x < 10))
if (!x || y)
if (flag_tb[i] && status)
if ((x != 1) && (x != 4) && (x != 10))
```

Non-compliant example

```
if (x > 0 && x < 10)
if (x != 1 && x != 4 && x != 10)
```

The objective of this rule is to write an expression that prevents confusion in understanding the order of precedence of each operand in && or ||. Its aim is to highlight the operation of each operand in && or || to improve the readability by enclosing the expression that contains an operator other than unary, postfix and cast operators with ( ). Another rule that may be considered is to enclose ! operation with ( ) because the order of precedence may be confusing to beginners.

[Related rules] R2.3.2 M1.5.2

M1.4.2 《Usage of parentheses to explicitly indicate operator precedence shall be defined.》

Preference guide	
Rule specification	Define

Compliant example

```
a = (b << 1) + c;
  or
a = b << (1 + c);
```

Non-compliant example

```
a = b << 1 + c;  // There is a possibility
                 // that operator precedence
                 // is misunderstood.
```

Operator precedence is difficult to capture. Therefore, set a rule as exemplified below to improve its readability.
If an expression contains multiple binary operators that differs in the order of operation priority, parentheses () shall be used to explicitly indicate the operator precedence, provided that the parentheses () may be omitted in four arithmetic operations.
To learn more about the operator precedence and its interpretation, refer to Chapter 10: Expressions of The C++ Programming Language C, Fourth Edition.

[Related rule] M1.5.1

M1.5 Explicitly describe the operations that are likely to cause misunderstanding when they are omitted.

M1.5.1 A function identifier (function name) shall only be used with either a preceding “&” or with a parenthesized parameter list, which may be empty.

Preference guide	
Rule specification	

Compliant example

```
void func(void);
void (*fp)(void) = &func;

if (func()) {
```

Non-compliant example

```
void func(void);
void (*fp)(void) = func;  // Non-compliant:No
&.

if (func) {  // Non-compliant:Address is
             // obtained rather than calling
             // the function.
            // It might be mistakenly
            // written as a function call
            // without arguments.
```

If a function name is written alone, it will be used to obtain the function address, and not for function call. This means that, for obtaining the function address, there is no need of placing `&` in front of the function name. However, the function name without a preceding `&`, in some cases, may be misunderstood that it is for a function call (for example, when using languages like Ada that write only the name to call a subprogram without arguments). By following the rule to add `&` when obtaining the function address, it will become easier to detect mistakes in function names written as they are with neither `&` nor `()`.
(for example, when using languages like Ada and Ruby that write only the name to call a subprogram without arguments).

[Related rule] M1.4.2

M1.5.2 The conditional expression in an if statement or loop shall explicitly state that the type is bool.

Preference guide	
Rule specification	

Compliant example

```
int x = 5;

if (x != 0) {
  …
}

class C {
  …
  bool isEmpty() const { // Function that
                         // returns bool
                         // type value.
    return …;
  }
  …
};

C obj;
…
if (!obj.isEmpty()) { // Compliant:Use a
                      // function that
                      // returns bool type
                      // value.
  …
}
```

Non-compliant example

```
int x = 5;

if (x) {
  …
}

class C {
  …
  operator bool() const { // Implicit
                          // conversion
                          // function.
    return …;
  }
  …
};

C obj;
…
if (obj) { // Non-compliant:Implicit type
           // conversion.
  …
}
```

The value in a conditional expression that is not bool type is implicitly converted into a bool value. Such implicit type conversion may cause unintended behaviors. Therefore, to clarify the intention of the program, the comparison shall not be omitted. Other preventive measures include the use of a function that returns an appropriate bool type value.

[Related rule] M1.4.1

M1.6 Use one area for one purpose.

M1.6.1 Variables shall be prepared for each purpose.

Preference guide	
Rule specification	

Compliant example

```
// Counter variable and work variable for
// replacement are different.
for (i = 0; i < MAX; i++) {
  data[i] = i;
}
if (min > max) {
  wk = max;
  max = min;
  min = wk;
}
```

Non-compliant example

```
// Same variable is used as counter variable
// and work variable for replacement.
for (i = 0; i < MAX; i++) {
  data[i] = i;
}
if (min > max) {
  i = max;
  max = min;
  min = i;
}
```

Reusing variables shall be avoided as it impairs readability and increases the risk of modification errors.

[Related rule] M1.2.1

M1.6.2

(1) `union`s shall not be used.

(2) If `union`s are used, the same members that are assigned values shall be referenced.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (2)
// If type is INT→i_var, CHAR→c_var[4],
struct stag {
int type;
union utag {
    char c_var[4];
    int i_var;
 } u_var;
};

struct stag  s_var;
…
int i;
…
if (s_var.type == INT) {
s_var.u_var.i_var = 1;
}
…
i = s_var.u_var.i_var;
```

Non-compliant example

```
Non-compliant example of (2)
// If type is INT→i_var, CHAR→c_var[4],
struct stag {
 int type;
 union utag {
  char c_var[4];
  int i_var;
 } u_var;
};

struct stag  s_var;

…
int i;
…
if (s_var.type == INT) {
s_var.u_var.c_var[0] = 0;
s_var.u_var.c_var[1] = 0;
s_var.u_var.c_var[2] = 0;
s_var.u_var.c_var[3] = 1;
}
…
i = s_var.u_var.i_var;
```

`union` allows the same memory space to be declared with areas of different sizes. However, unexpected behavior may occur, depending on the compiler that differs in the way the bits overlap among members. If `union` is going to be used, follow rule (2).

[Related rule] R2.1.3

M1.7 Do not reuse names.

M1.7.1

The following rules shall be followed to ensure name uniqueness:

1. **An identifier declared in an inner scope shall not hide an indentifier declared in an outer scope.**
2. **A `typedef` name (including qualification, if any) shall be a unique identifier.**
3. **A tag name, `union` name or enumeration name (including qualification, if any) shall all be a unique identifier.**
4. **No object or function identifier with static storage duration should be reused.**
5. **No identifier in one category should have the same spelling as an identifier in another category.**

Preference guide	○
Rule specification	

The program will become easier to read by using unique names within the program, except for cases like automatic variables where the scope is limited and cases like polymorphism where the name is redefined.

In C++ language, in addition to the scope defined by file and block, names are classified into the following three categories:

1. Label
2. Class name, union name, enumeration name
3. Other identifiers

Note, however, that for rule 2 above, there is a restriction that all type names must be different, except for cases where the same type is defined. See below for examples:
(Excerpts from Annex C, C.1.6, ISO/IEC 14882:2011.)

```
typedef struct name1 { /*…*/ } name1; // valid C and C++
struct name { /*…*/ };
typedef int name; // valid C, invalid C++
```

The language specification allows assigning the same name to identifiers in different categories, but the objective of the above rules is to improve the readability of the program by restricting the reuse of same names.

[Related rule] M4.3.1

M1.7.2 Names of functions, variables and macros in the standard library shall not be redefined or reused. In addition, these macro names shall not be undefined.

Preference guide	○
Rule specification	

Compliant example

```
#include <string.h>
void *my_memcpy(void *arg1, const void
                *arg2, size_t size) {
  …
}
```

Non-compliant example

```
#undef NULL
#define NULL ((void *)0)

#include <string.h>
void *memcpy(void *arg1, const void *arg2,
             size_t size) {
  …
}
```

Redefining the names of functions, variables and macros defined in the standard library degrades the readability of the program.

[Related rules] M1.7.3 M4.3.1

M1.7.3 Names (variables) that start with an underscore shall not be defined.

Preference guide	○
Rule specification	

Compliant example

—

Non-compliant example

```
int _Max1;     // Reserved.
int __max2;    // Reserved.
int _max3;     // Reserved.

struct S {
  int _mem1;   // Not reserved, but shall not
               // be used.
};
```

In the language standard, the following names are defined as reserved:

(1) Name that starts with an underscore and is followed by either an upper case letter or another underscore;

Examples: `_Abc`, `__abc`

(2) All names that start with an underscore;

These names are reserved for variables and functions with file scope as well as for tags.

When the reserved names are redefined, the behavior of the compiler will not be guaranteed. Names that start with an underscore and are followed by a lower case letter are actually not reserved for use outside the file scope. But to set a rule that is easy to remember, the current rule is defined to restrict the use of all names starting with an underscore.

[Related rules] M1.7.2 M4.3.1

M1.8 Do not use language specifications that are likely to cause misunderstanding.

M1.8.1 The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

Preference guide	
Rule specification	

Compliant example

```
a = *p;
p++;
// p has been counted up regardless of the
// content pointed by p.
if ((MIN < a) && (a < MAX)) {
    ...
}

volatile int *io_port = ... ; // Address for
                              // memory mapped
                              // I/O
int io_result = *io_port;
// I/O is processed, regardless of the
// conditions of the if statement.
if ((x != 0) && (io_result > 0)) {
    ...
}
```

Non-compliant example

```
// Whether p is counted up or not depends on
// whether the content pointed by p is
// smaller than MIN or greater than or equal
// to MIN.
if ((MIN < *p) && (*p++ < MAX)) {
    ...
}

volatile int *io_port = ... ; // Address for
                              // memory
                              // mapped I/O
// Whether I/O is processed or not varies,
// depending on the conditions of the if
// statement.
if ((x != 0) && (*io_port > 0)) {
    ...
}
```

The right-hand side of `&&` or `||` operators may not be executed, depending on the result of the condition of their left-hand side. Take, for example, an expression with a side effect of incrementing. It this expression is written on the right-hand side, whether the increment is executed or not will be difficult to understand, because it depends on the condition of the left-hand side. Therefore, expressions with side effects shall not be described on the right-hand side of `&&` or `||` operators.

[Related rules] R3.6.1 R3.6.2

M1.8.2

(1) Macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
(2) Macros shall only expand to a guard for prevention of redundant inclusion of the header file, a type qualifier, or a storage class specifier.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1)
#define START 0x0410
#define STOP 0x0401

Compliant example of (2)
#ifndef MYHEADER_H
#define MYHEADER_H
```

Non-compliant example

```
#define BIGIN {
#define END }
#define LOOP_STAT for (; ;) {
#define LOOP_END }
```

Macro definitions can be leveraged to make the code look like it is written in a language other than C++, or greatly reduce the amount of code. However, using macros for such purposes will degrade readability. The use of macros, therefore, should be limited to only where coding and modification errors can be prevented.
For do-while-zero, see MISRA C:2004.

M1.8.3

`#line` shall not be used, unless it is automatically generated by a tool.

Preference guide	
Rule specification	

`#line` serves as the means to intentionally modify file names or line numbers of warning or error messages output from the compiler. It is provided under the assumption that code is generated by tools, and is not intended to be used directly by the programmers.

M1.8.4 Sequences of three or more characters starting with ?? and alternative tokens shall not be used.

Preference guide	
Rule specification	

Compliant example

```
s = "abc\?\?(x)";
```

Non-compliant example

```
s = "abc??(x)";  // Compilers that can process
                 // trigraph sequences
                 // interpret this as "abc(x)".
```

C++ language standard defines trigraph sequences and alternative tokens, assuming that there may be cases where some characters cannot be used for coding, depending on the environment used for development.

The following nine three-character patterns, known as trigraph sequences:

```
??=  ??(   ??/   ??)   ??'   ??<   ??!   ??>   ??-
```

can be replaced respectively with the following corresponding single-character counterparts at the beginning of the preprocessor:

```
#    [     \     ]     ^     {     |     }     ~
```

The following character patterns that are defined as alternative tokens:

```
<%     %>     <:     :>     %:     %:%:    and   bitor   or     xor   compl
bitand      and_eq          or_eq          xor_eq          not         not_eq
```

are treated respectively as equivalent of the following corresponding character patterns:

```
{      }      [      ]      #      ##      &&     |       ||     ^     ~
&           &=              |=             ^=              !           !=
```

Since trigraph sequences and alternative tokens are not frequently used, many compilers support them as an optional feature.

M1.8.5 A sequence starting with zero (0) that is two or more digits long shall not be used as a constant.

Preference guide	
Rule specification	

Compliant example

```
a = 0;
b = 8;
c = 100;
```

Non-compliant example

```
a = 000;
b = 010;
c = 100;
```

Constants starting with zero (0) are interpreted as octal. No zero (0) can be added in front of decimal numbers to align their digits for the purpose of appearance (i.e.: zero padding is not allowed).

[Related rule] M1.2.2

M1.8.6 && || , (comma) and & operators shall not be overloaded.

Preference guide	●
Rule specification	

Compliant example

```
class C {
public:
  bool and_cc(C const &, C const &);
    // Compliant:
  …
};

C c;
and_cc(c, func());  // func is always called.
```

Non-compliant example

```
class C {
public:
  bool operator &&(C const &, C const &);
    // Non-compliant:
  …
};

C c;
c && func();  // There are times when func
              // is not called.
```

When overloading an operator, be sure to maintain the meaning of the operator defined in the programming language (== operator is for equality, + operator is for addition, and so on).
C++ language specification defines that short-circuit evaluation shall be executed on the right-hand side of `&&` and `||` (meaning that it may not be executed depending on the left-hand value) and that for `,` , the operands shall be evaluated in the order of left-hand to right-hand side. But because the behavior of the overloaded operator is defined as a function, it is difficult to meet the requirement to evaluate in left-to-right order. At the same time, if `&` which is an operator that seeks for the object address is overloaded, the function that will be called will vary, depending on whether the type is undefined or not.

【Reference material for those wanting to know more in detail about this rule】

- C++ Coding Standards Item 30

M1.8.7 `explicit` specifier shall be used for conversion function.

Preference guide	●
Rule specification	

Compliant example

```
class C {
public:
  explict operator int() { ... } // Compliant:
  …                              // explicit specifier
                                 // is used.
};
C c:
int x = c;         // Compile error willoccur.
int y = int(c);    // Compile error will not
                   // occur.
```

Non-compliant example

```
class C {
public:
  operator int() { ... } // Non-compliant:
  …                      // explicit specifier
                         // is not used.
};
C c:
int x = c;   // Conversion function (operator
             // int) is called implicitly.
```

Conversion function may be called implicitly in the code where type conversion is necessary. By using explicit specifier that has been additionally introduced in C++11 in the code where conversion function is implicitly called, error can be processed when the code is compiled. Since the type conversion will be explicitly written, the readability of the code will improve. `explicit` specifier cannot be used if the compiler in use does not support C++11. In that case, there would be a need to define the conversion function and call it explicitly to prevent the implicit conversion when type conversion is written.

Example:
```
class C {
public:
  int as_int() {…} // Function that performs conversion
}
C c;
int x = C.as_int();
```

【Reference materials for those wanting to know more in detail about this rule】

- More Effective C++ Item 5 (In case of using C++03)

[Related rules] M1.8.8 E1.1.7

M1.8.8 Single-parameter constructor shall have an `explicit` specifier.

Preference guide	○
Rule specification	

Compliant example

```
class C {
public:
  explicit C(int n) {   // Compliant:explicit
                        // specifier
    …
  }
  …
};

void func1(const X &x) { … }
void func2(void) {
  C x(0);
  func1(x);   // Compile OK
  func1(0);   // Compile error
}
```

Non-compliant example

```
class C {
public:
  C(int n) {   // Non-compliant:Single-
               // parameter constructor
               // without specifier

    …
  }
  …
};

void func1(const X &x) { … }
void func2(void) {
  func1(0);   // Constructor "C::C(int)" is
              // implicitly called.

}
```

Single-parameter constructor may be called implicitly at where type conversion is necessary. When the constructor has an `explicit` specifier, the description to call the constructor implicitly can be handled as a compile error. And since the constructor call is explicitly described, the readability will improve.

【Reference material for those wanting to know more in detail about this rule】

- More Effective C++ Item 5

[Related rules] M1.8.7 E1.1.7

M1.9 When writing in an unconventional style, explicitly state its intention.

M1.9.1 If statements that do nothing need to be intentionally described, comments or empty macros shall be used to make them noticeable.

Preference guide	○
Rule specification	

Compliant example

```
for (;;) {
  // Waiting for interrupt.
}

#define NO_STATEMENT
i = COUNT;
while ((--i) > 0) {
  NO_STATEMENT;
}
```

Non-compliant example

```
for (;;) {
}

i = COUNT;
while ((--i) > 0);
```

[Related rule] M1.1.1

M1.9.2 《The unified style of writing infinite loops shall be defined.》

Preference guide	○
Rule specification	Define

Define the unified style of writing infinite loops by selecting from one of the following:

- Write the infinite loops uniformly as `for(;;)`;.
- Write the infinite loops uniformly as `while(1)`;.
- Use the macro defined for the infinite loop.

M1.10 Do not embed magic numbers.

M1.10.1 A meaningful constant shall be used after defining it as a constant with a name.

Preference guide	○
Rule specification	

Compliant example

```
(1)Compliant example when const is used
const int MAXCNT = 8;
if (cnt == MAXCNT) { // Compliant: const
                     // qualifier is used.

(2)Compliant example when constexpr is used
constexpr int MAXCNT = 8;
if (cnt == MAXCNT) { // Compliant: constexpr
                     // qualifier is used.
```

Non-compliant example

```
if (cnt == 8) { // Non-compliant: Literal is
                // used.
  …
```

The meaning of the constant can be stated explicitly by defining it as a constant with a name. In addition, when modifying a program where the same constant is used in multiple places, modification errors can be prevented much more easily if this same constant is defined with a name, because then, there will only be a need to modify the definition of one constant. The constant with a name is defined by using `const` qualifier or enumeration type (enum). Macro can also provide the same effect as constant with a name, but it is better to use constant with a name rather than macro. For one reason, constant with a name can have a scope, whereas macro cannot. Furthermore, constant with a name defined by `const` or `enum` can be referenced at the time of symbolic debugging unlike the macro name, and therefore makes debugging easier.
C++11 and later versions of C++ language standards allow the constant to be defined by `constexpr` qualifier when the code is compiled. Therefore, use `constexpr` qualifier to define the constant with a name when using a compiler that supports C++11 and later versions of C++ language standards.
However, for referencing the data size, use `sizeof` instead of constant with a name.

[Related rule] M2.2.4

M1.11 Explicitly state the area attributes.

M1.11.1 Read-only areas shall be declared as `const` type.

Preference guide	○
Rule specification	

Compliant example

```
const volatile int read_only_mem;
   // Read-only memory.
const int constant_data = 10;
   // Read-only data that does not require
   // memory allocation.
void func(const char *arg, int n) {
   // Only reads the contents pointed by arg.
  int i;
  for (i = 0; i < n; i++) {
    put(*arg++);
  }
}
void CLS::func(const char *arg, int n) const {
   // Data members are not updated.
  …
}
```

Non-compliant example

```
int read_only_mem;  // Read-only memory.
int constant_data = 10;  // Read-only data
                         // that does not
                         // require memory
                         // allocation.
void func(char *arg, int n) { // Only reads
                              // the contents
                              // pointed by
                              // arg.
  int i;
  for (i = 0; i < n; i++) {
    put(*arg++);
  }
}
void CLS::func(char *arg, int n) {
             // Data members are not updated.
  …
}
```

Variables that are only referenced and not modified can be indicated clearly that they must not be modified by declaring them with const type. In addition, the object code size may become smaller through optimization by the compiler. Therefore, read-only variables should be declared with const type. Memories that are only referenced by a particular program and modified by other execution units should be declared with const volatile type, so that the compiler can detect the error caused by this program when it updates them by mistake.
Furthermore, declaration with `const` allows the function interfaces to be explicitly stated, when the area indicated by the parameter is only referenced in function processing, or when member function only reference the data members.

M1.11.2 Areas that may be updated by other execution units shall be declared as `volatile`.

Preference guide	○
Rule specification	

Compliant example

```
volatile int x = 1;
while (x == 1) {
  // x is not modified within the loop and is
  // modified by other execution units.
}
```

Non-compliant example

```
int x = 1;
while (x == 1) {
  // x is not modified within the loop and is
  // modified by other execution units.
}
```

Areas qualified as `volatile` prohibit the compiler from optimizing them. Prohibition of optimization means that executable object is generated strictly to every description, including even those considered logically as unnecessary of processing. Suppose there is a description `x;` that has no meaning logically except for only referencing variable `x`. If it is not qualified as `volatile`, the compiler will normally ignore such statement and will not generate an executable object. Whereas, if it is qualified as `volatile`, the compiler will generate an executable object that only references variable `x` (loads it to the register). This description can be assumed to have meaning in indicating the interface to IO registers (mapped to the memory) that are reset when the memory is read. Embedded software has IO registers for controlling hardware, that should be qualified as `volatile` when considered appropriate, based on their characteristics.

M1.11.3 《Rules for variable declaration and definition for ROMization shall be defined.》

Preference guide	
Rule specification	Define

Compliant example

```
constexpr int x = 100;  // Allocate to ROM.
```

Non-compliant example

```
int x = 100;
```

Variables qualified as constexpr can be allocated to ROMization target areas. For example, when developing a program where ROMization is applied, qualify the read-only variables as `constexpr`, and specify the name of the section to which these variables are allocated by, such as, `#pragma`.
In case of using C++03, `const` qualification shall be used.

M1.12 Correctly describe the statements even if they are not compiled.

M1.12.1 Correct code shall be described even if it is going to be deleted by the preprocessor.

Preference guide	
Rule specification	

Compliant example

```
#if 0
  //
# endif

#if 0
  …
#else
  int var;
#endif

#if 0
  // I don't know
#endif
```

Non-compliant example

```
#if 0
  /*
# endif

#if 0
  …
#else1
  int var;
#endif

#if 0
  I don't know
#endif
```

[Related rule] M1.1.2

Maintainability M2 Write in a style that can prevent modification errors.

One of the patterns that allows bugs to slip into a program easily is when other bugs are created by mistake while fixing detected bugs. Especially if it has been a while since the source code was written or if an engineer other than the creator modifies the source code, unexpected misunderstanding may occur.
Efforts to reduce such modification errors as much as possible are strongly desired.

Maintainability M2.1	**Clarify the grouping of structured data and blocks.**
Maintainability M2.2	**Localize access ranges and related data.**
Maintainability M2.3	**Commonalize the code used to do the same process.**

M2.1 Clarify the grouping of structured data and blocks.

M2.1.1 If arrays and structures are initialized with values other than 0, their structural form shall be indicated by using braces '{ }'. Data shall be described without any omission, except when all values are 0.

Preference guide	○
Rule specification	

Compliant example

```
int arr1[2][3] = {{0, 1, 2}, {3, 4, 5}};
int arr2[3] = {1, 1, 0};
```

Non-compliant example

```
int arr1[2][3] = {0, 1, 2, 3, 4, 5};
int arr2[3] = {1, 1};
```

In initialization of arrays and structures, at least a pair of braces '{ }' is required, but in this case, it is difficult to see how the initial values are assigned. Therefore, create blocks according to the structure, and fully describe the initial values without omitting any.

[Related rule] M4.5.3

M2.1.2 The body of `if`, `else if`, `else`, `while`, `do`, `for`, and `switch` statements shall be enclosed into blocks.

Preference guide	
Rule specification	

適合例

```
if (x == 1) {
  func();
}
```

Non-compliant example

```
if (x == 1)
  func();
```

If there is only one statement that is controlled by, such as, an if statement, there is no need to enclose this statement into a block. However, when the program is modified and this single statement is changed into multiple statements, there is a possibility of forgetting to enclose these multiple statements into a block. To prevent such modification errors, enclose the body of each controlled statement into a block.

M2.2 Localize access ranges and related data.

M2.2.1 Variables used only in one function shall be declared within the function.

Preference guide	●
Rule specification	

Compliant example

```
void func1(void) {
  static int x = 0;
  if (x != 0) {   // Refer to the value in the
                  // immediately preceding
                  // call.
    x++;
  }
  …
}
void func2(void) {
  int y = 0;      // Initialize each time.
  …
}
```

Non-compliant example

```
int x = 0;  // x is accessed only from
func1.
int y = 0;  // y is accessed only from
func2.
void func1(void) {
  if (x != 0) {  // Refer to the value in the
                 // immediately preceding
                 // call.
    x++;
  }
  …
}
void func2(void) {
  y = 0;    // Initialize each time.
  …
}
```

To declare variables in functions, it is sometimes effective to declare them with `static` storage class specifiers. The following positive effects can be expected by using `static` storage class specifiers:

- Static memory space is reserved and this space remains valid until the end of the program. (Generally without the `static` storage class specifier, stack memory is used and remains valid until the end of the function.)
- Initialization occurs only once after starting the program, and when a function is called more than once, the value assigned in the immediately preceding call is retained.

Therefore, among the variables accessed only within a function, the variables with values that are retained even after the function terminates should be declared with `static` storage class specifiers.
In addition, declaring a large memory space for an automatic variable may cause stack overflow. When there is such risk, one preventive measure is to use the `static` storage class specifier to reserve static memory space even if the values do not need to be retained after the function terminates. However, when using the `static` storage class specifier for such purpose, its intention should be explicitly stated by, such as, comments (to prevent potential misunderstanding that the `static` storage class specifier has been used by mistake).

[Related rule] M2.2.2

M2.2.2 Variables accessed by several functions defined in the same file shall be declared by either one of the following methods so that they will become inaccessible from other files:

(1) By declaring these variables outside of function, using the `static` storage class specifiers;

(2) By declaring these variables in unnamed namespace.

Preference guide	○
Rule specification	Choose

Compliant example

```
Compliant example of (1)
// x is not accessed from other files.
static int x;
void func1() {
  …
  x = 0;
  …
}
void func2() {
  …
  if (x == 0) {
    x++;
  }
  …
}

Compliant example of (2)
// x is not accessed from other files.
namespace {
  int x; // Compliant: It cannot be accessed
         // from other files because it is
         // declared by using the unnamed
         // namespace.
}
void func1() {
  …
  x = 0;
  …
}

void func2() {
  …
  if (x == 0) {
    x++;
  }
  …
}
```

Non-compliant example

```
// x is not accessed from other files.
int x;  // Non-compliant:x can be accessed
        // from other files.
void func1() {
  …
  x = 0;
  …
}
void func2() {
  …
  if (x == 0) {
    x++;
  }
  …
}
```

The fewer the global variables, the higher the readability of the entire program becomes. To prevent the number of global variables from increasing, either declare the local variables with `static` storage class specifiers or declare them in unnamed namespace.

[Related rules] M2.2.1 M2.2.3

M2.2.3 Functions called from only the functions defined in the same file shall be declared by either one of the following methods so that they will not be called from other files: (1) By declaring these functions as `static` functions; (2) By declaring these functions in unnamed namespace.

Preference guide	○
Rule specification	Choose

Compliant example

```
Compliant example of (1)
// func1 is not called from functions in
// other files.
static void func1(void) {
  …
}
void func2() {
  func1();
}

Compliant example of (2)
// func1 is not called from functions in
// other files.
namespace {
  void func1 (void);
// Compliant: It cannot be accessed from
// other files because it is declared by
// using the unnamed namespace.
}
void func2() {
  …
  func1();
  …
}
```

Non-compliant example

```
// func1 is not called from functions in
// other files.
void func1(void) {
// Non-compliant: func1 can be called from
// functions in other files.
  …
}
void func2() {
  …
  func1();
  …
}
```

The fewer the global functions, the higher the readability of the entire program becomes. To prevent the number of global functions from increasing, either declare the local functions with `static` storage class specifiers or declare them in unnamed namespace.

[Related rule] M2.2.2

M2.2.4 enum shall be used when defining related constants.

Preference guide	
Rule specification	

Compliant example

```
enum ecounty {
  ENGLAND, FRANCE, …
};
enum eweek {
  SUNDAY, MONDAY, …
};

enum ecounty country;
enum eweek   day;
…

if (country == ENGLAND) {
if (day == MONDAY) {
if (country == SUNDAY) { // Can be checked
                         // with tool.
// country = SUNDAY; will be a compile error.
```

Non-compliant example

```
const int ENGLAND = 0;
const int FRANCE = 1;
const int SUNDAY = 0;
const int MONDAY = 1;
int country, day;
…
if (country == ENGLAND) {
if (day == MONDAY) {
if (country == SUNDAY) {
             // Cannot be checked with tool.
country = SUNDAY ; // country = SUNDAY; will
                   // not be a compile error.
```

To define the constants that are related like a set, use the enumeration type. By defining related constants as `enum` type, and using this type instead of `#define`, `const` qualifier or `constexpr` qualifier, the use of incorrect values can be prevented.
Moreover, `enum` constants defined by `enum` declaration will be the names processed by the compiler. The names processed by the compiler are easier to debug, because they can be referenced during symbolic debugging.
By using `enum class` or `enum struct` that have been added in C++11, name conflict can be prevented since the enumerator will be included in the scope of `enum class` (`struct`). Mistakes can also be prevented because the compiler will process the comparison of enumerators with different `enum` type as an error.

[Related rules] M1.10.1 P1.3.2

M2.2.5 Data members shall be `private`.

Preference guide	○
Rule specification	

Compliant example

```
class CLS {
public:
  CLS() : i(10) { }

  void setValue(int x) {i = x;}
  int getValue() {return i:}
    …
private:
  int i;  // Compliant:Data members are
          // private.
;

int main() {
  CLS c;
  c.setValue(30);  // Data members cannot be
                   // accessed directly from
                   // outside.
}
```

Non-compliant example

```
class CLS {
public:
  CLS() : i (10) { }

  int i;  // Non-compliant:Data members are
          // public.
};

int main() {
  CLS c;
  c.i = 30;  // Data members can be accessed
             // directly from outside.

}
```

Specify the data members as `private` to prevent them from being changed directly from outside. By doing so, when the class is expanded or changes are made to the class, the scope of impact of such expansion or changes can be limited to within the class, and the maintainability of the code can be improved. For the same reason, do not make the pointer of a data member specified as `private`, the return value of the member function, without `const`-qualifying it.
As an exception, design the program carefully if the resources are going to be open to the public, due to such reasons as compatibility with other systems.

【Reference material for those wanting to know more in detail about this rule】

- C++ Coding Standards Item 42

M2.3 Commonalize the code used to do the same process.

M2.3.1 A constructor shall be used for object initialization processed in the same way.

Preference guide	○
Rule specification	

Compliant example

```
class C {
private:
   int x;
public:
   // Compliant: One constructor is used to
   // do the same processes.
   C() : C(0) {}
   C(int v) {
      x = f(v);
   }
   ...
};
```

Non-compliant example

```
class C {
private:
   int x;
public:
   // Non-compliant: Multiple constructors
   // are used to do the same processes.
   C() {
      x = f(0);
   }
   C(int v) {
      x = f(v);
   }
   ...
};
```

When a code used to do the same process exists in multiple locations and there is a need to correct the process of that code, there is a risk of inconsistency being caused by correcting the process of that code in only some locations and failing to correct the process of that code in other locations. This problem can be prevented by commonalzing the code used to do the same process. In C++11, a feature to call a different constructor from a constructor (delegating constructor) has been added. By using this new feature, the same initialization process can be grouped together without preparing a separate initialization function.

[Related rule] R1.4.1

Maintainability M3 - Write programs simply.

From the standpoint of software maintainability, there is just no better software than those created from simply written programs.

C++ language enables the structuring of software by, such as, dividing the program into separate source files and functions. Structured programming that represents program structure through three forms: sequence, selection and repetition, is also one of the applicable techniques to write simple software programs. Writing simple software descriptions through effective use of software structuring is strongly desired. Moreover, particular attention should also be given to writing styles applied to describe, such as, iteration processing, assignment and operations, as some may make the program difficult to maintain.

Maintainability M3.1	**Do structured programming.**
Maintainability M3.2	**Limit the number of side effects per statement to one.**
Maintainability M3.3	**Write expressions that differ in purpose separately.**
Maintainability M3.4	**Do not use complicated pointer operation.**
Maintainability M3.5	**Do not use complicated class structure.**

M3.1 Do structured programming.

M3.1.1 For any iteration statement, there shall be at most one `break` statement or `goto` statement used `for` loop termination.

Preference guide	
Rule specification	

Compliant example

```
Compliant example of (1)
for (i = 0; Loop iteration condition; i++) {
  Iterated processing 1;
  if (Termination condition 1 || Termination
                                   condition 2) {
    break;
  }
  Iterated processing 2;
}

Compliant example of (2)
end = 0;
for (i = 0; Loop iteration condition &&
!end; i++) {
  Iterated processing 1;
  if (Termination condition 1 || Termination
                                   condition 2) {
    end = 1;
  } else {
    Iterated processing 2;
  }
}
```

Non-compliant example

```
for (i = 0; Loop iteration condition; i++) {
  Iterated processing 1;
  if (Termination condition 1) {
    break;
  }
  if (Termination condition 2) {
    break;
  }
  Iterated processing 2;
}
```

These rules are to help improve the readability of the logical structure of the program. When a flag becomes necessary only for eliminating the `break` statement, sometimes it is better not to prepare the flag, but rather to use a `break` statement. (Be careful, however, when using an end flag like in the case shown above as Compliant example of (2), because it may impair the readability of the program.)

M3.1.2

(1) The `goto` statement shall not be used.

(2) When using a `goto` statement, the destination to jump to shall be the label declared after the `goto` statement that is within the block enclosing the `goto` statement.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1), (2)
for (i = 0; Loop iteration condition; i++) {
  Iterated processing; // goto is not
                       // included.
}

Compliant example of (2)
{
  if (err != 0) {
    goto ERR_RET;
  }
...
ERR_RET:
  end_proc();
  return err;
}
```

Non-compliant example

```
Non-compliant example of (1), (2)
  i = 0;
LOOP:
  Iterated processing ;
  i++;
  if (Loop iteration condition) {
    goto LOOP;
  }
```

These rules are to help improve the readability of the logical structure of the program. The purpose is not to eliminate all the `goto` statements. The important point is to eliminate unnecessary `goto` statements to prevent the readability of the program from being impaired (i.e., not being able to read it straightforwardly from top to bottom). In some cases, the readability can actually be improved by writing `goto` statements. Therefore, when programming, keep in mind how simply the logic can be expressed.
For example, `goto` statement can be useful to make the program simple, such as, when it is used to jump to error processing or exit from multiple loops.

*M3.1.3 is deleted and vacant. (see the table in Appendix)

M3.1.4

(1) Each `case` clause and `default` clause in a `switch` statement shall always end with a `break` statement.
(2) If a `case` clause or `default` clause in a `switch` statement is not going to be ended with a `break` statement, 《a project-specific comment shall be defined》 and that comment shall instead be inserted.

Preference guide	○
Rule specification	Choose/ Define

Compliant example

```
Compliant example of (1), (2)
switch (week) {
case A :
  code = MON;
  break;
case B :
  code = TUE;
  break;
case C :
  code = WED;
  break;
default:
  code = ELSE;
  break;
}

Compliant example of (2)
dd = 0;
switch (status) {
case A :
  dd++;
  // FALL THROUGH
case B :
```

Non-compliant example

```
Non-compliant example of (1), (2)
// No matter what the value of week is, the
// code will be ELSE.
// ==> Bug
switch (week) {
case A :
  code = MON;
case B :
  code = TUE;
case C :
  code = WED;
default:
  code = ELSE;
}
// This is a case where processing of case B
// can be continued after dd++, but it is
// non-compliant not only to (1) but also to
// (2) because there is no comment.
dd = 0;
switch (status) {
case A :
  dd++;
case B :
```

One of the typical examples of coding error is caused by forgetting to write the `break` statement in a `switch` statement. To prevent it, avoid writing a `case` statement without the `break` statement unnecessarily. If the code is intended to continue processing to the next `case` without the `break` statement, always insert a comment to explicitly indicate that the absence of the `break` statement is not a problem. Define what kind of comment to insert in such `case` in the coding convention. As one example, `// FALL THROUGH` is a comment that is frequently used.

[Related rule] R3.5.2

M3.1.5

(1) A function shall end with one `return` statement.
(2) A `return` statement to return in the middle of processing shall be written only in case of recovery from abnormality.

Preference guide	
Rule specification	Choose

These rules are to help improve the readability of the program logic. When a program has many entry or exit points, they will not only impair the readability of the program, but also increase the number of break points for debugging.

M3.2 Limit the number of side effects per statement to one.

M3.2.1

(1) Comma expressions shall not be used.

(2) Comma expressions shall not be used, other than in expressions for initializing or updating in for statements.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1), (2)
a = 1;
b = 1;

j = 10;
for (i = 0; i < 10; i++) {
  …
  j--;
}

Compliant example of (2)
for (i = 0, j = 10; i < 10; i++, j--) {
  …
}
```

Non-compliant example

```
Non-compliant example of (1), (2)
a = 1, b = 1;

Non-compliant example of (1)
for (i = 0, j = 10; i < 10; i++, j--) {
  …
}
```

In general, the use of comma expressions impairs readability. However, the readability may sometimes improve by using comma expressions to bind all the pre-loop operations as one set and all the loop-end operations as another set, and placing them respectively in expressions for initializing and updating in for statements.

[Related rule] M3.3.1

M3.2.2 Multiple assignments shall not be written in one statement, except when the same value is assigned to multiple variables.

Preference guide	○
Rule specification	

Compliant example

```
x = y = 0;
```

Non-compliant example

```
y = (x += 1) + 2;
y = (a++) + (b++);
```

Assignments include the compound assignments (+= -=, etc) beside the simple assignment (=). Multiple assignments may be written in one statement, but since they impair readability, one statement should contain only one assignment.
However, “commonly used conventional descriptions” shown below do not impair readability in many cases. They may be treated as exceptions of this rule.

```
c = *p++;
*p++ = *q++;
```

M3.3 Write expressions that differ in purpose separately.

M3.3.1 The three expressions of a for statement shall be concerned only with loop control.

Preference guide	
Rule specification	

Compliant example

```
for (i = 0; i < MAX; i++) {
  …
  j++;
}
```

Non-compliant example

```
for (i = 0; i < MAX; i++, j++) {
  …
}
```

[Related rules] M3.2.1 M3.3.2

M3.3.2 Numeric variables being used within a for loop `for` iteration counting shall not be modified in the body of the loop.

Preference guide	
Rule specification	

Compliant example

```
for (i = 0; i < MAX; i++) {
  …
}
```

Non-compliant example

```
for (i = 0; i < MAX; ) {
  …
  i++;
}
```

[Related rule] M3.3.1

M3.3.3 (1) Assignment operators shall not be used in expressions to examine true or false. (2) Assignment operators shall not be used in expressions to examine true or false, except for conventionally used notations.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1), (2)
p = top_p;
if (p != NULL) {
  …
}

Compliant example of (1)
c = *p++;
while (c != '\0') {
  …
  c = *p++;
}
```

Non-compliant example

```
Non-compliant example of (1), (2)
if (p = top_p) {
  …
}

Non-compliant example of (1)
while (c = *p++) {
  …
}
Since this is an expression used
conventionally, it is compliant to (2).
(However, be careful of its usage, because
its readability depends on the programmer’s
coding skills.)
```

The expressions for evaluating true or false are as follows:

`if (` *expression* `)`	`for ( ;` *expression* `; )`	`while (` *expression* `)`
`(` *expression* `) ? :`	*expression* `&&` *expression*	*expression* `||` *expression*

M3.4 Do not use complicated pointer operation.

M3.4.1 Three or more pointer indirections shall not be used.

Preference guide	
Rule specification	

Compliant example

```
int **p;
typedef char **strptr_t;
strptr_t q;
```

Non-compliant example

```
int ***p;
typedef char **strptr_t;
strptr_t *q;
```

Since it is difficult to understand the changes in the pointer values in three or more levels, multiple pointer indirections impair maintainability.

M3.5 Do not use complicated class structure.

M3.5.1 Virtual inheritance and non-virtual inheritance shall not be mixed in an accessible base class in the same hierarchical structure.

Preference guide	○
Rule specification	

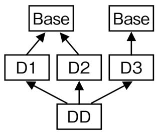
Compliant example

```
class Base { … };
class D1 : public virtual Base { … };
class D2 : public virtual Base { … };
class D3 : public virtual Base { … };
class DD : D1, D2, D3 { … }
    // Compliant:D1, D2, D3 are all
    // virtually inherited from Base.
```

Non-compliant example

```
class Base { … };
class D1 : public virtual Base { … };
class D2 : public virtual Base { … };
class D3 : public Base { … };
    // Non-virtual inheritance.
class DD : public D1, D2, D3 { … }
    // Non-compliant:D1 and D2 are virtually
    // inherited, but D3 is non-virtually
    // inherited.
```

When virtual inheritance and non-virtual inheritance are mixed in a declaration, the behavior will be based on non-virtual inheritance. For example, if classes D1, D2 and D3 are inherited from the same base class, where inheritance of classes D1 and D2 are virtual while the inheritance of class D3 is non-virtual, D1 and D2 will contain the data members of the same base class, whereas D3 will contain the data members of a different base class. Avoid creating such complicated structure that is difficult to understand.



[Related rule] E1.1.9

Maintainability M4 Write in a unified style.

Recently, developing programs under the shared efforts of multiple programmers has become a widely accepted approach in software projects. If these programmers apply different coding styles to write their assigned portion of the source code, the reviewers or other programmers may later face difficulty checking what each programmer has written. Moreover, if the naming of variables, information to be described in a file, and the order to describe the information, among others, are not uniform, unexpected misunderstanding or errors may arise from such inconsistencies. This is why writing the source code as much as possible according to a unified coding style in a single project or within the organization is often said to be desirable.

Maintainability M4.1	Unify the coding styles.
Maintainability M4.2	Unify the style of writing comments.
Maintainability M4.3	Unify the naming convention.
Maintainability M4.4	Unify the contents to be described in a file and the order of describing them.
Maintainability M4.5	Unify the style of writing declarations.
Maintainability M4.6	Unify the style of writing null pointers.
Maintainability M4.7	Unify the style of writing preprocessor directives.
Maintainability M4.8	Unify the style of writing overloads.

M4.1 Unify the coding styles.

M4.1.1 《Conventions regarding the style of using, such as, the braces'{ }', indentation and space shall be defined.》

Preference guide	○
Preference guide	Define

To make the code easier to read, it is important to unify the coding style applied in the project.
When defining a new style convention to be followed in the project, the recommended approach would be to select from already existing coding styles. Existing coding styles have been developed from various schools, and many programmers create their programs based on any one or more of these pre-established styles. One of the benefits of selecting from these existing coding styles is that the format can be easily specified by the format commands available in editors and other tools. If no coding style is clearly specified in the existing project, the recommendation would be to define a coding convention that matches most closely with the current source code.
What is most important in deciding on the style convention is not in "deciding what kind of style" to define, but is in "defining a unified style to be followed".
Explained below are the set of style-related items to be defined:

1) Position of braces '{ }'

Unify the position to place the braces '`{ }`' so that the beginning and end of a block will become easier to read (see **Representative styles**).

2) Indentation

Indentation makes a group of declarations and operations easier to read. For unified use of indentation, define the following:

- Whether to use space or tab for indentation;
- If space is used, how many space characters are used for one indent? If tab is used, how many characters are set for each tab?

3) How to use spacing

Spacing makes the code easier to read. For example, define the following rules:

- Add a space before and after a binary or ternary operator, except for the following operators:
  `[ ]`, `->`, `.` (period), `,` (comma operator):
- Do not add a space between a unary operator and its operand.

By applying these rules, coding errors that are attributable to compound assignment operators will become easier to detect.

[Examples]

```
x=-1;   // Intended to write x-=1, but made a mistake => difficult to distinguish
x =- 1; // Intended to write x-=1, but made a mistake => easy to distinguish
```

Besides those stated above, the following rules are also defined in some cases:

● Add a space after a comma (except for commas for parameters in macro definitions)

● Add a space before the left parenthesis '(' enclosing control expressions in, such as, if and for. Do not add a space before the left parenthesis '(' of a function call.

This rule makes it easier to identify function calls.

Until C++98, nested template that ended with consecutive '>' had to be differentiated from right-shift operator by inserting a blank space in between '>'. The use of spacing in this manner is no longer mandatory from C++11. Whether to enter a blank space in between '>' should therefore be discussed in each project and set as a local coding rule. Setting such local coding rule is also preferable in case there is a need to consider the compatibility with the language specifications defined in C++03.

[Example]

```
list<list<int>>  list0; // No blank space ⇒ Not compatible with C++03
list<list<int> > list1; // With a blank space ⇒ Compatible with C++03
```

4) Position to place a new line character for line continuation

When an expression becomes lengthy and extends beyond the length of an easily readable line, a new line character shall be placed at an appropriate position. In placing a new line character, the recommended approach is to apply either one of the following two methods. What is important is to write the continuation line after indenting.

[Method 1] Write an operator at the end of the line.

Example:

```
x = var1 + var2 + var3 + var4 +
    var5 + var6 + var7 + var8 + var9;
if (var1 == var2 &&
      var3 == var4)
```

[Method 2] Write an operator at the beginning of the continuation line.

Example:

```
x = var1 + var2 + var3 + var4
    + var5 + var6 + var7 + var8 + var9;
if (var1 == var2
      && var3 == var4)
```

●Representative styles

1) K&R style

This is a coding style used in "The C Programming Language" (widely known as K&R). "K&R" used as the acronym of this book derives from the initials of the two authors. In the K&R style, the braces '{ }' and indentation are placed in the positions described below:

•Position of braces: Place the braces '{ }' for function definitions at the beginning of a new line indented to the same column as the preceding line above. Place the braces '{ }' for others (including structures and control statements, such as, if, for and while) on the same line without continuing to a new line (see Example of K&R style).

•Indentation: 1 tab. In the first edition of "The C Programming Language", the width of a tab was set to 5 spaces, but in the second edition (ANSI compliant), the number of spaces is set to 4.

2) BSD style

This is a description style adopted by Eric Allman who wrote many BSD utilities. It is also called the Allman style. In the BSD style, the braces '{ }' and indentation are placed in the positions described below:

•Position of braces: Start all the function definitions, if, for and while, etc, from a newline and place the braces '{ }' at the column aligned with the beginning of the previous line (see Example of BSD style).

•Indentation: 8 spaces. 4 spaces are also common.

3) GNU style

This is a coding style for writing GNU packages. It is defined in "GNU Coding Standards" written by Richard Stallman and volunteers of the GNU project. In the GNU style, the braces '{ }' and indentation are placed in the positions described below:

•Position of braces: Start all the function definitions, if, for and while, etc, from a new line. Place the braces '{ }' for function definitions at column 0, and braces '{ }' for others after indenting 2 spaces (see Example of GNU style).

•Indentation: 2 spaces. Indent 2 spaces for both the braces '{ }' and their body.

4) Stroustrup style (The C++ Programming Language, 3rd ed.)

●**Position of braces:** Start the braces '{ }' for all the function definitions from a new line, and place them at the column aligned with the previous line. However, if the body of the member function can be written in one line, place the braces '{ }' also on the same line without continuing to a new line. For others (including struct, class, if, for, while, try, and catch), place the braces '{ }' on the same line without continuing to a new line.

●**Indentation:** 1 tab (4 columns). Do not indent when writing access specifiers (private, protected, public).

```
(1) Example of K&R style:
void func(int arg1)
{ // Write the { of a function on a new
  // line.
  // Indent is 1 tab.
    if (arg1) {
      …
    }
    …
}
```

```
(2)  Example of BSD style:
void
func(int arg1)
{ // Write the { of a function on a new
  // line.
    if (arg1)
    {
      …
    }
      …
}
```

```
(3) Example of GNU style:
void
func(int arg1)
{ // Write the { of a function on a new
  // line at column 0.
  if (arg1)
    {
      …
    }
  …
}
```

```
(4)Example of Stroustrip style:
void func(int arg1)
{   // Write the { of a function on a new
    // line.
    if (arg1) {
      …
    }
}
class CLS {
private:
    int m1;
    int m2;
public:
    CLS() { … }
    CLS(int a, int b)
      : m1(a),
        m2(b)
  { … }
  virtual void show(void) const{ … }
};
```

M4.1.2 C++ style casting shall be used, provided that casting to `void` shall be allowed.

Preference guide	
Rule specification	

Compliant example

```
int show(char *str);
...
double d = 3.14;
int i = static_cast<int>(d);
              // Compliant: C++ style casting.
const int *cip = &i;
int *ip = const_cast<int *>(cip);
              // Compliant: C++ style casting.
long l = reinterpret_cast<long>(ip);
              // Compliant: C++ style casting.
(void) show("abc"); // Compliant
```

Non-compliant example

```
int show(char *str);
...
double d = 3.14;
int i = (int)d;
          // Non-compliant: C style casting.
const int *cip = &i;
int *ip = (int *)cip;
          // Non-compliant: C style casting.
long l = (long)ip;
          // Non-compliant: C style casting.
```

C++ style casting is better for readability than C style casting because C++ style expresses the intention of cast more clearly than C style.
However, casting to void can be used because it can state explicitly that the return value of function call is to be ignored.

[Related rules] R2.7.1 R2.7.4

M4.2 Unify the style of writing comments.

M4.2.1 《Convention regarding the style of writing file header comments, function header comments, end of line comments, block comments and copyright shall be defined.》

Preference guide	○
Rule specification	Define

Writing good comments makes the program easier to read. To improve the readability further, a unified style of writing is necessary.

There are document generation tools that create documents for maintenance and examination from the source code. When utilizing such tools, they can be most effectively used by writing in a style that conforms to their specifications. In general, when the explanation of the variables and functions are described according to certain comment conventions, the document generation tools enable these descriptions to be extracted from the source code and reflected in the generated documents. Therefore, it is important to examine the specifications of these tools and define the comment conventions accordingly.

Presented below are some established styles of writing comments that have been extracted from existing coding conventions and related literature.

● Representative styles of writing comments

1) Indian Hill coding conventions

The following comment rules are described in Indian Hill C Style and Coding Standards [14]:

- **Block comments**

  Comments that describe data structures, algorithms, etc., should be in block comment form with the opening / in column 1, a * in column 2 before each line of comment text, and the closing */ in columns 2-3.

  (Note that `grep ^ . \ *` will catch all block comments in the file.)

  Example:
  ```
  /* Write a comment
   * Write a comment
   */
  ```

- **Position of comments**

  - Block comments inside a function

    Should be tabbed over to the same tab setting as the code that they describe.

  - End-of-line comments

    Very short comments may appear on the same line as the code they describe, but should be tabbed over far enough to separate them from the statements. If more than one short comment appears in a block of code they should all be tabbed to the same tab setting.

2) GNU coding standards

The following comment rules are described in the GNU Coding Standards [24]:

- **Language** English.
- **Position and contents**
  - At the beginning of the program
    Write a comment that briefly explains what the program does at the beginning of every program.
  - Function
    Write comments that provide the following information for each function.
    What the function does, explanation of parameters (values, meaning, usage), return value
  - #endif
    Except for short conditions that are not nested, add comments to explicitly state the conditions at the end of line of every #endif.
  - Notation for tools
    Place two spaces at the end of each comment sentence.

3) "The Practices of Programming"

The following comment rules are described in "The Practices of Programming" [16], co-authored by Brian W. Kernighan and Rob Pike:

- **Position** Describe comments for functions and global data.
- **Other practices**
  - Don't belabor the obvious.
  - Don't contradict the code.
  - Clarify, don't confuse

4) Others

- Set a policy on using /**/ comments and // comments.
  Example 1: Use // for end-of-line comments and /**/ for block comments.
  Example 2: Use only //, because with /**/, there is a risk of forgetting to close /* with */ at the end of comment.
  Example 3: Do not use /* or // in a comment, provided that // may be used in a // comment.

- Describe the copyright notice in the comment.

● Define the comment for the case clauses without break.

Example:

```
switch (status) {
case CASE1:
  Processing;
  // FALL THROUGH
case CASE2:
  …
```

● Define the comment for no processing (in case the else condition does not occur).

Example:

```
if (Condition 1) {
  Processing;
} else if (Condition 2) {
  Processing;
} else {
  // NOT REACHED
}
```

● Line-splicing shall not be used in // comments. (MISRA C:2012 R3.2)

M4.3 Unify the naming convention.

M4.3.1 《Convention for naming external variables and internal variables shall be defined.》

Preference guide	○
Rule specification	Define

See **Rules on naming convention** below.

[Related rules] M1.7.1 M1.7.2 M1.7.3 M4.3.2 P1.2.1

M4.3.2 《Convention for naming files shall be defined.》

Preference guide	○
Rule specification	Define

See **Rules on naming convention** below.

[Related rules] M4.3.1 P1.2.1

● Rules on naming convention

Readability of programs is greatly affected by naming. There are various methods of naming, but the important points are to be consistent and to make the names easy to understand.

For naming, the following items shall be defined:

- Guideline for names in general;
- How to name files (including folders and directories);
- How to name globally and locally
- How to name macros, etc.

Presented below are some naming guidelines and rules introduced in existing coding conventions and related literature. They are useful as reference when creating a project-specific naming convention newly. If no naming convention is explicitly defined in the existing project, a naming convention that is closest to the current source code shall be defined.

●Representative naming conventions

1) Indian Hill coding conventions

- Names with leading and trailing underscores are reserved for system purposes and should not be used.
- #define constant names should be all in CAPS.
- enum member names should either have the initial character or all the characters capitalized.
- It is best to avoid names that differ only in case, like foo and FOO.
- Global names should have a common prefix for identifying the module they belong to.
- A file name should be eight characters or less (excluding the extension), starting with an alphabetic character and followed by alphanumeric characters.
- File names that are the same as library header filenames should be avoided.

Overall		● Names with leading and trailing underscores should not be used. ● It is best to avoid names that differ only in case. Example: foo and Foo
Variable names, function names	**Global**	A prefix for identifying the module should be added.
	Local	Nothing in particular
Others		● Macro names should be all in CAPS. Example: `#define` *MACRO* ● enum member names should either have the initial character or all the characters capitalized.

2) GNU coding standards

- Don't choose terse names—instead, look for names that give useful information about the meaning of the variable or function. Names should be English.
- Use underscores to separate words in a name.
- Stick to lower case; reserve upper case for macros and enum constants, and for name-prefixes that follow a uniform convention.

Overall		● Use underscores to separate words in a name. Example: get_name ● Stick to lower case; reserve upper case for macros and enum constants, and for name-prefixes that follow a uniform convention.
Variable names, function names	**Global**	Don't choose terse names—instead, look for names that give useful information about their meaning in English.
	Local	Nothing in particular
Others		● Macro names should be all in CAPS. Example: `#define` *MACRO* ● enum member names should be all in CAPS.

3) The Practice of Programming

- Use descriptive names for globals, short names for locals.
- Give related things related names that show their relationship and highlight their difference.
- Function names should be based on active verbs, perhaps followed by nouns.

Overall		Give related things related names that show their relationship
Variable names, function names	**Global**	Use descriptive names.
	Local	Use short names.
Others		Function names should be based on active verbs, perhaps followed by nouns.

4) JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM [19]

- All words in an identifier will be separated by the '_' character.
- User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.
- Identifiers will not begin with the underscore character '_'.
- Identifiers will not differ by:
  - Only a mixture of case
  - The presence/absence of the underscore character
  - The interchange of the letter 'O', with the number '0' or the letter 'D'
  - The interchange of the letter 'I', with the number '1' or the letter 'l'
  - The interchange of the letter 'S' with the number '5'
  - The interchange of the letter 'Z' with the number '2'
  - The interchange of the letter 'n' with the letter 'h'.
- All acronyms in an identifier will be composed of uppercase letters. Example: `RGB_colors`
- The first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.
  Example: `class Base { … } ; enum RGB_colors { red, green, blue } ;`
- All letters contained in function and variable names will be composed entirely of lowercase letters.
  Example: `example_function_name`
- Identifiers for constant and enumerator values shall be lowercase.
  Example: `const int max_colors = 255 ; enum RGB_colors { red, green, blue } ;`
- Header files will always have a file name extension of ".h". Implementation files will always have a file name extension of ".cpp".

5) Others

● How to separate a name:	A name that consists of multiple words should be either separated with underscore or delimited by capitalizing the first letter of the separate word. Determine which style to adopt.
● Hungarian notation:	There is a notation called Hungarian notation that explicitly indicates the type of variable.
● How to name files:	Give a name with a prefix, for example, that expresses the subsystem.

[Related rules] M4.3.1 P1.2.1

M4.4 Unify the contents to be described in a file and the order of describing them.

M4.4.1 《The descriptive contents of header files (declarations, definitions, etc) and the order they are described in shall be defined.》

Preference guide	○
Rule specification	Define

Items commonly used in a program shall be described in header files to prevent the risk of modification errors when they are scattered in different places. Header files should contain macro definitions, declarations of const constants, definitions of class, template class, structure, union and enumeration types, typedef declarations, external variable declarations, function prototype declarations and template function definition that are commonly used in multiple source files.
For example, they should be described in the following order:

Example 1: Header file used for class definition:

1) File header comment
2) Inclusion of system headers (in the order of C libraries, C++ libraries, others)
3) Inclusion of user-defined headers
4) Class definition

Example 2: Header file used for purposes other than class definition:

1) File header comment
2) Inclusion of system headers (in the order of C libraries, C++ libraries, others)
3) Inclusion of user-defined headers
4) Declarations of `const` constants
5) `enum` definitions
6) `typedef` definitions (type definition of basic types, such as, `int` and `char`)
7) `extern` variable declarations
8) Function prototype declarations
9) Template function definitions
10) inline functions

M4.4.2 《The descriptive contents of source files (declarations, definitions, etc) and the order they are described in shall be defined.》

Preference guide	○
Rule specification	Define

In a source file, definitions of variables, functions and template functions, definitions or declarations of macros, const constants, classes, template classes, structures, unions and enumerations, and types (typedef types) used only in the individual source file should be described.
For example, they should be described in the following order:

Example 1: Source file used for defining member functions

1) File header comment
2) Inclusion of system headers
3) Inclusion of user-defined headers
4) Member function definitions

*Regarding 2) and 3), do not include unnecessary items.
*Regarding 4), define all the member functions of a class in the same file.

Example 2: Source file used for defining other than member functions

1) File header comment
2) Inclusion of system headers
3) Inclusion of user-defined headers
4) `#define` macros used only in this file
5) Declarations of const constants that are used only in this file
6) `#define` function macros used only in this file
7) `typedef` definitions used only in this file
8) `enum` tag definitions used only in this file
9) `struct/union` tag definitions used only in this file
10) Class definition (including template class) used only in this file
11) Declarations of static variables shared in this file
12) static function declarations
13) Variable definitions
14) Template function definitions
15) Function definitions

*Regarding 2) and 3), do not include unnecessary items.
*Avoid describing 4) through 10) as much as possible.

M4.4.3 To use or define external variables or functions (except for functions used only in the file), the header file describing their declarations shall be included.

Preference guide	○
Rule specification	

Compliant example

```
Compliant example
--- my_inc.h ---
extern int x;
int func(int);

--------------
#include "my_inc.h"
int x;
int func(int in) {
  …
```

Non-compliant example

```
// Declarations of variable x and function
// func are missing.
int x;
int func(int in) {
  …
```

To ensure that declarations and definitions are consistent, the declarations should be described in the header file, and that header file should be included.

M4.4.6 Header files shall be descriptively capable of handling redundant inclusions. 《The descriptive method to achieve this capability shall be defined.》

Preference guide	○
Rule specification	Define

Compliant example

```
--- myheader.h ---
#ifndef MYHEADER_H
#define MYHEADER_H
Contents of the header file
#endif // MYHEADER_H
```

Non-compliant example

```
--- myheader.h ---
void func(void);
// end of file
```

The descriptive contents of header files should be organized to avoid redundant inclusions. However, there are cases when redundant inclusions become unavoidable. To prepare for such cases, header files should be written in such a way that will make them possible of handling multiple inclusions.

As an example, the following may be defined as the rule for writing header files that are capable of handling redundant inclusions:

Example of the rule: `#ifndef` macro that judges whether the header has already been included or not shall be written at the beginning of the header file, so that the descriptions that follow will not be compiled in subsequent inclusions. In this case, the macro name should be the same as the header file name but replacing all the lowercase letters to uppercase letters, and the period '.' to underscore '_'.

*M4.4.4 and M4.4.5 are deleted in C++ Language edition and are vacant. (see the table in Appendix)

M4.4.7 Using directive and using declaration of the namespace shall not be written before #include in the source file or in the header file, except in case of writing using declaration in class scope or function scope.

Preference guide	○
Rule specification	

Compliant example

```
Compliant example of (1)
--- file1.h ---
namespace NS1 {
  int func(double x) {return x + 1;}
}
--- file2.h ---
namespace NS2 {
  int g1() {return NS1::func(1);}
     // Compliant:Scope resolution operator
     // is used.

}
--- file3.h ---
inline int func(int x) {return x;}
namespace NS2 {
  int g2() {return func(1);}
}
--------------
#include "file1.h"
#include "file2.h"
#include "file3.h"  // ::func(int) is called
                    // by func(1) of file3.h.

--------------
#include "file1.h"
#include "file3.h"  // ::func(int) is called
                    // by func(1) of file3.h.

#include "file2.h"

Compliant example of (2)
class Base {
public:
  std::size_t size() const {return n;}
protected:
  std::size_t n;
}
class Derived : private Base {
public:
  using Base::size;
     // Compliant: The access level from
     // Derived of Base::size() is changed
     // to public.
protected:
  using Base::n;
     // Compliant:The access level from
     // Derived of Base::size() is changed
     // to protected.
```

Non-compliant example

```
--- file1.h ---
namespace NS1 {
  inline int func(double x) {return x + 1;}
}
--- file2.h ---
namespace NS2 {
  using NS1::func;
  // Non-compliant:using declaration of
  // namespace.

  inline int g1() {return func(1);}
}
--- file3.h ---
inline int func(int x) {return x;}
namespace NS2 {
  inline int g2() {return func(1);}
}
--------------
#include "file1.h"
#include "file2.h"
#include "file3.h"  // fNS1::func(double)
                    // is called by func(1)
                    // of file3.h.


--------------
#include "file1.h"
#include "file3.h"  // ::func(int) is called
                    // by func(1) of file3.h.

#include "file2.h"
```

If using directive or using declaration of the namespace is written before #include in the source file or in the header file, the names written later will be interpreted differently. In such case, either state the name explicitly by using the scope resolution operator or write the using declaration of the namespace after `#include` in the source file.

M4.4.8 《The order of writing the class members shall be defined.》

Preference guide	
Rule specification	Define

Define the order of writing the class members by taking account of the access levels.
A typical example of this order would be as follows.

Example:

Write in the following order.

```
public
protected
private
```

M4.5 Unify the style of writing declarations.

M4.5.1

(1) In a function prototype declaration, all the parameters shall not be named (types only.)

(2) In a function prototype declaration, all the parameters shall be named. In addition, the types of the parameters, their names and the type of the return value shall be literally the same as those of the function definition.

Preference guide	○
Rule specification	Choose

Compliant example

```
Compliant example of (1)
int func1(int, int);

int func1(int x, int y) {
  // Process the function.
}

Compliant example of (2)
int func1(int x, int y);
int func2(float x, int y);

int func1(int x, int y) {
  // Process the function.
}

int func2(float x, int y)
  // Process the function.
}

void func(int a[4])
{
  // Process the function.(size of a = 4)
}
```

Non-compliant example

```
Non-compliant example of (1) , (2)
int func1(int x, int y);
int func2(float x, int y);

int func1(int y, int x) {
  // The parameter name differs from the
  // name in the prototype declaration.
  // Process the function.
}

typedef int INT;
int func2(float x, INT y) {
  // The type of y is not literally the same
  // as the type in the prototype
  // declaration.
  // Process the function.
}

void func(int a[])  // size not specified
{
  // Process the function.(size of a assumed 4)
}
```

In a function prototype declaration, parameter names can be omitted, but describing appropriate parameter names is useful as function interface information. When describing parameter names, use the same name as in the definition to avoid unnecessary confusion. As for the parameter type name, make it literally the same as the function definition to make the code easier to read.
As a principle, any parameter that is not used in the function must be deleted from the function. However, there are some cases like in the example below, where it is acceptable to only omit unused parameter names from the function definition.

Example:

```
int func(int x, int, int z) ; // Second parameter is not used.
int func(int x, int, int z) {
  // Process the function.
}
```

Moreover, if the parameter is an array of specific size, it is desirable to specify the number of its elements.

[Related rule] M1.1.1

M4.5.2 Definition of class or enumeration and function declaration shall be performed separately.

Preference guide	
Rule specification	

Compliant example

```
struct TAG {
  int mem1;
  int mem2;
};
struct TAG x;
```

Non-compliant example

```
struct TAG  {
  int mem1;
  int mem2;
} x;
```

M4.5.3 (1) ‘,’ shall not be placed before the last ‘}’ in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators.
(2) ‘,’ shall not be placed before the last ‘}’ in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators. However, placing ‘,’ before the last ‘}’ in the list of initial values for array initialization is allowed.

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (1)
struct tag data[] = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}  // There is no comma after the
             // last element.
};

Compliant example of (2)
struct tag data[] = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},  // There is a comma after the
              // last element.
};
```

Non-compliant example

```
Non-compliant example of (1) , (2)
struct tag x = {1, 2,};
// Not clear whether there are only two
// members or whether there are three or
// more.
```

The usage of comma in descriptions for initializing multiple data is generally divided into two schools of coding rules. One school follows the tradition of not placing a comma after the last initial value in order to indicate the end of initialization explicitly. Another school follows the tradition of placing a comma at the end by considering the easiness of adding or deleting initial values. Decide on which rule to follow by weighing the importance of the usage of comma for such descriptions in your specific cases.

[Related rule] M2.1.1

M4.6 Unify the style of writing null pointers.

M4.6.1

(1) `nullptr` shall be used for the null pointer.
(2) `0` shall be used for the `null` pointer. `NULL` shall not be used in any case.
(3) `NULL` shall be used for the null pointer. `NULL` shall not be used for anything other than the null pointer.

Preference guide	○
Rule specification	Choose

Compliant example

```
Compliant example of (1)
char *p;

p = nullptr;

Compliant example of (2)
char *p;
int dat[10];

p = 0;
dat[0] = 0;

Compliant example of (3)
char *p;
int dat[10];

p = NULL;
dat[0] = 0;
```

Non-compliant example

```
Compliant example of (2)
char *p;
int dat[10];

p = NULL;
dat[0] = NULL;

Compliant example of (3)
char *p;
int dat[10];

p = 0;
dat[0] = NULL;
```

`0` and `NULL` have been conventionally used to express the null pointer, but the expression of the null pointer varies, depending on the execution environment. Since nullptr has been newly introduced in C++11, `nullptr` should be used for null pointer constant.
The use of 0 or NULL to express the null pointer should be limited to only the code developed according to the language specifications defined in C++03 or older versions that will not be revised according to the language specifications defined in later versions. In such code, either `0` or `NULL` should be used to express the null pointer throughout the entire code, and a local rule should be set to refrain from using the other to express the null pointer in the same code.

【Reference materials for those wanting to know more in detail about this rule】
- Effective Modern C++ Item 8

M4.7 Unify the style of writing preprocessor directives.

M4.7.1 The body and parameters of a macro that includes operators shall be enclosed with parentheses `()`.

Preference guide	○
Rule specification	

Compliant example

```
#define M_SAMPLE(a, b) ((a) + (b))
```

Non-compliant example

```
#define M_SAMPLE(a, b) a + b
```

If the body and parameters of a macro are not enclosed with parentheses ( ), there is a risk of bug being produced when the operations are not performed in the expected order, since the operation order depends on the order of precedence of operators that come next to the macro after expanding the macro and the operators in the macro.

M4.7.2 `#else`, `#elif` or `#endif` that corresponds to `#ifdef`, `#ifndef` or `#if` shall be described in the same file, and 《their correspondence relationship shall be clearly stated with a comment defined in the project》.

Preference guide	○
Rule specification	Define

Compliant example

```
#ifdef AAA
  Process when AAA is defined
  …
#else  // not AAA
  Process when AAA is not defined
  …
#endif  // end AAA
```

Non-compliant example

```
#ifdef AAA
  Process when AAA is defined
  …
#else
  Process when AAA is not defined
 …
#endif
```

If `#else` or `#endif` is described in a distant location or nested in a partitioned process by macros, such as, `#ifdef`, their correspondence becomes difficult to understand. Therefore, add a project-defined comment to `#else` or `#endif` that corresponds with, such as, `#ifdef` to make their correspondence easier to understand.

[Related rules] M1.1.1 M1.1.2

M4.7.3 To check whether a macro name has already been defined or not with `#if` or `#elif`, `defined(`*macro_name*`)` or `defined` *macro_name* shall be used.

Preference guide	●
Rule specification	

Compliant example

```
#if defined(AAA)
  …
#endif
#if defined BBB
  …
#endif
```

Non-compliant example

```
#if AAA
  …
#endif
#define DD(x)   defined(x)
#if DD(BBB)
  …
#endif
```

`#if` *macro_name* does not determine whether a macro is defined or not. In case of `#if AAA` for example, the judgment will be false not only when macro `AAA` is not defined, but also when the value of macro `AAA` is 0 (zero). Therefore, `defined` should be used to check whether a macro is defined or not.
For writing `defined`, use either `defined(`*macro_name*`)` or `defined` *macro_name*. Other ways of writing defined should be avoided since they are implementation-dependent (i.e. some compilers may process them as error).

*M4.7.4 is deleted and vacant. (see the table in Appendix)

M4.7.5 Macros shall not be #define'd or #undef'd within a block.

Preference guide	
Rule specification	

Compliant example

```
#define AAA 0
#define BBB 1
#define CCC 2
struct stag {
  int mem1;
  char *mem2;
};
```

Non-compliant example

```
// Members with restriction on assignable
// values exist.
struct stag {
  int mem1;  // The following values are
             // assignable:
#define AAA 0
#define BBB 1
#define CCC 2
  char *mem2;
};
```

In general, macro definitions (`#define`) are all described together at the beginning of the file. If they are scattered in various parts of the file, for example, by describing them in blocks, they will become difficult to read. Moreover, cancellation of definitions (`#undef`) within a block will also degrade the readability. Also note that, unlike the scope of variables, macro definitions are valid only up to the end of the file. The description below shows how the program in the above non-compliant example can be rewritten to make it compliant:

```
enum etag {AAA, BBB, CCC};
struct stag {
  enum etag mem1;
  char *mem2;
};
```

[Related rule] M4.7.6

M4.7.6 #undef shall not be used.

Preference guide	
Rule specification	

#undef can change the state of #define'd macro name to undefined. But the use of #undef involves the risk of degrading the readability, because the interpretation of #undef may differ, depending on where the macro name is referred to.

[Related rule] M4.7.5

M4.7.7 The controlling expression of a #if or #elif preprocessing directive shall be evaluated to 0 or 1.【MISRA C:2012 R20.8】

Preference guide	
Rule specification	

Compliant example

```
#define TRUE   1
#define FALSE  0
#if TRUE
     ...
#if defined(AAA)
     ...
#if VERSION == 2
     ...
#if 0   // Invalidated due to ～.
```

Non-compliant example

```
#define ABC 2
#if ABC
     ...
```

In case of #if or #elif controlling expression, true or false is evaluated by the controlling expression. Therefore, the controlling expression should be described in a way that would make it easy to evaluate true or false, thus making the program easy to read.

M4.8 Unify the style of writing overloads.

M4.8.1 Operator overload shall be defined according to the original meaning of the operator and together with other operators in the same category.

Preference guide	●
Rule specification	

Compliant example

```
class CLS {
public:
  bool operator != (const CLS &c) const {
    // Compliant: With == operator
    // definition.
    if (cls_i == c.cls_i) {
      return false;
    }
    return true;
  }
  bool operator == (const CLS &c) const {
    // Compliant: With != operator definition.
    if (cls_i != c.cls_i) {
      return false;
    }
    return true;
  }

  CLS(int i) : cls_i(i) { }
private:
  int cls_i;
};
```

Non-compliant example

```
class CLS {
public:
  bool operator != (const CLS &c) const {
    // Non-compliant: Without == operator
    // definition.
    if (cls_i == c.cls_i) {
      return false;
    }
    return true;
  }

  CLS(int i) : cls_i(i) { }
private:
  int cls_i;
};
```

When overloading the operator, be sure to maintain the meaning of the operator defined in the programming language (== operator is for equality, + operator is for addition, and so on). When the definition of the overloaded operator is going to deviate from the original meaning, define the overloaded operator as a function. When defining an overloaded operator, also define other overloaded operators that the users (in each project) consider using as a set of operators in the same category.

Category	Operators in the same category
Unary plus and minus	unary + unary −
Additive operation	prefix++ postfix++ prefix−− postfix−− + − += −=
Multiplicative operation	* / % *= /= %=
Bit shift	<< >> <<= >>=
Relational operation	< > <= >=
Equality operation	== !=
Bit operation	unary ~ & ^ | &= ^= |=

M4.8.2 When defining operators new and delete of a class, all the standard forms shall be defined.

Preference guide	
Rule specification	

Compliant example

```
class CLS {
public:
// Operators new and delete in single-object form
  static void *operator new(std::size_t);
                                 // Exception is thrown.
  static void *operator new(std::size_t,
     const std::nothrow_t&) noexcept;
                             // Exception is not thrown.
  static void operator delete(void *)
    noexcept;                // Exception is not thrown.
  static void operator delete(void *,
    const std::nothrow_t&) noexcept;
                             // Exception is not thrown.
// Operators new and delete in array form
  static void *operator new[](std::size_t);
                                 // Exception is thrown.
  static void *operator new[](std::size_t ,
    const std::nothrow_t&) noexcept;
                             // Exception is not thrown.
  static void operator  delete[](void *)
   noexcept;                 // Exception is not thrown.
  static void operator delete[](void *,
    const std::nothrow_t&) noexcept;
                             // Exception is not thrown.
// Operators new and delete in placement form
  static void *operator new (std::size_t, void *)
    noexcept;                // Exception is not thrown.
  static void *operator new[](std::size_t, void *)
    noexcept;                // Exception is not thrown.
  static void operator delete (void *, void *)
   noexcept;                 // Exception is not thrown.
  static void operator delete[](void *,void *)
    noexcept;                // Exception is not thrown.
…
};

[According to C++03]
class CLS {
public:
// Operators new and delete in single-object form
  static void *operator new(std::size_t)
    throw(std::bad_alloc);      // Exception is thrown.
  static void *operator new(std::size_t,
    const std::nothrow_t&) throw();
                             // Exception is not thrown.
  static void operator delete(void *)
    throw();                 // Exception is not thrown.
  static void operator delete(void *,
    const std::nothrow_t&) throw();
                             // Exception is not thrown.
// Operators new and delete in array form
  static void *operator new[](std::size_t)
    throw(std::bad_alloc);      // Exception is thrown.
```

Non-compliant example

```
class CLS {
public:
// Non-compliant:Global new and delete
// operators are hidden.
   static void *operator new(std::size_t) ;
     // Exception is thrown.
   static void operator delete(void *) noexcept ;
     // Exception is not thrown.
…
};

[In case of C++03]
class CLS {
public:
// Non-compliant:Global new and delete
// operators are hidden.
   static void *operator new(std::size_t)
     throw(std::bad_alloc);
     // Exception is thrown.
   static void operator delete(void *) throw();
     // Exception is not thrown.
…
};
```

```
  static void *operator new[](std::size_t ,
    const std::nothrow_t&) throw();
                              // Exception is not thrown.
  static void operator delete[](void *)
    throw();                  // Exception is not thrown.
  static void operator delete[](void *,
    const std::nothrow_t&) throw();
                              // Exception is not thrown.
// Operators new and delete in placement form
  static void *operator new (std::size_t, void *)
   throw();                   // Exception is not thrown.
  static void *operator new[](std::size_t, void *)
    throw();                  // Exception is not thrown.
  static void operator delete (void *, void *)
    throw();                  // Exception is not thrown.
  static void operator delete[](void *,void *)
   throw();                   // Exception is not thrown.
…
```

When defining operators `new` and `delete` of a class, define all the following standard forms:

- Operators new and delete in single-object form (that are not in array form)
  - That throw exceptions
  - That do not throw exceptions
- Operators `new` and `delete` in array form
  - That throw exceptions
  - That do not throw exceptions
- Operators `new` and `delete` in placement form
  - That do not throw exceptions

When user-defined `new` and `delete` operators are provided in a class, global `new` and `delete` operators of the same name provided for that class by default will be hidden. Therefore, unless all the standard forms are defined, the compiler may not be able to find the correct new and delete operators actually required for execution.
Operators `new` and `delete` in array form and operators `new` and `delete` in single object form have different names(*). Therefore, even when user-defined operators `new` and `delete` in single object form, for example, are provided in a class, the default global `new` and `delete` operators in array form will not be hidden. However, from the standpoint of handling consistency, when creating user-defined `new` and `delete` operators, define both in single object form and array form.
* Name difference
- Single-object form: `operator new` `operator delete`
- Array form: `operator new[]` `operator delete[]`

【Reference materials for those wanting to know more in detail about this rule】
- C++ Coding Standards Item 46

[Related rule] M5.2.1

M4.8.3 Operators new and delete shall be defined in pairs.

Preference guide	●
Rule specification	

Compliant example

```
class CLS {
public:
  static void *operator new(std::size_t);
  static void operator delete(void) noexcept;
  static void *operator
    new(std::size_t, std::ostream&);
  // Compliant: Corresponding operator delete
  // does not exist.
  static void operator
    delete(void*, std::stream&) noexcept;
  ...
};

[In case of C++03]
class CLS {
public:
  static void *operator new(std::size_t)
                    throw(std::bad_alloc) ;
  static void operator delete(void) throw();
  static void *operator
    new(std::size_t, std::ostream&)
            throw(std::bad_alloc) ;
  // Compliant: Corresponding operator delete
  // exists.
  static void operator
    delete(void*, std::stream&) throw() ;
  ...
};
```

Non-compliant example

```
class CLS {
public:
  static void *operator new(std::size_t);
  static void *operator delete(void*)
                                noexcept;
  static void *operator
    new(std::size_t, std::ostream&);
  // Non-compliant: Corresponding operator
  // delete does not exist.
  ...
};

[In case of C++03]
class CLS {
public:
  static void *operator new(std::size_t)
                 throw(std::bad_alloc);
  static void *operator delete(void*) throw() ;
  static void *operator
    new(std::size_t, std::ostream&)
            throw(std::bad_alloc);
  // Non-compliant: Corresponding operator
  // delete does not exist.
  ...
};
```

When user-defined `new` operator is provided, that means that the default memory allocation method is changed. Normally, when the memory allocation method is changed, the method of releasing the memory is also changed accordingly. Therefore, when user-defined `new` operator is provided, also define the corresponding operator `delete` as a pair.
Moreover, when defining operator new in a class, it must be done carefully. If only operator `new` is defined without defining the corresponding operator `delete`, memory leak may occur because operator `delete` will not be called if the constructor fails after successful memory allocation for the object. This risk is attributable to C++ language specification that prevents the compiler from inserting the code to call operator `delete` when it cannot find operator `delete` of the same form as operator `new`. In other words, when an operator `new` of a non-standard form is defined(*), the compiler will not insert the code to call operator `delete`, unless operator `delete` of the same form as user-defined operator `new` is defined.

* The first parameter of `new` is fixed as `std::size_t`. But from second parameter onward, the users can freely define any parameter.

【Reference materials for those wanting to know more in detail about this rule】

- C++ Coding Standards Item 45
- Effective C++ Item 52

[Related rule] M5.2.1

Maintainability

M5 Write in a style that makes testing easy.

One of the essential tasks in embedded software development is to check the behaviors (through testing). However, with recent complex embedded software, it is becoming increasingly challenging to fulfill this task when faced with difficulties caused by, such as, bugs and malfunctions detected during tests that cannot be reproduced. Therefore, when writing the source code, it is desirable to be more conscious of writing in a style that will make the root cause analysis easy to perform when problem arises. Moreover, particular attention must also be given to descriptions that involve, such as, the use of dynamic memory, by keeping in mind the risk of memory leak, among other points of concern.

Maintainability M5.1	**Write in a style that makes it easy to investigate the causes of problems when they occur.**
Maintainability M5.2	**Be careful when using dynamic memory allocations.**

M5.1 Write in a style that makes it easy to investigate the causes of problems when they occur.

M5.1.1 《The rules for writing the code for setting debug options and for recording logs in release modules shall be defined.》

Preference guide	○
Rule specification	Define

Besides implementing the specified functionalities correctly, a good program requires coding that also take account of the easiness to debug and investigate into the causes of problems when they occur. Descriptions that make investigation of problems easy to conduct can be achieved by writing descriptions for debugging that are not reflected in the release modules and descriptions for outputting logs after release that are reflected in the release modules. Explained below are the points to take into consideration when determining the rules to be followed in writing each of these descriptions.

●Descriptions for debugging

Descriptions for debugging, including print statements used during program development, need to be written as isolated descriptions that are not reflected in the release module. Explained below are two ways of writing the descriptions for debugging: (a) by isolating the debug descriptions using macro definitions; and (b) by using assert macros for debugging purpose.

(a) Using macro definitions to isolate debug descriptions

Use the macro definitions to identify the code parts to be compiled so that the debug descriptions are not reflected in the provided release module. Strings, such as, “`DEBUG`” and “`MODULEA_DEBUG`” that contain “`DEBUG`” as part of the name are commonly used as those macro names.

Example of rule definition:

`#ifdef DEBUG` shall be used to isolate the debug code. (`DEBUG` macro shall be specified at compile time.)

[Code example]

```
#ifdef DEBUG
fprintf(stderr, "var1 = %dn", var1);
#endif
```

The following macro definitions can also be used.

Example of rule definition:

`#ifdef DEBUG` shall be used to isolate the debug code. (`DEBUG` macro shall be specified at compile time.) In addition, the following macro shall be used to output debug information.

```
DEBUG_PRINT(str); // Output str to standard output
```

Since this macro is defined in the common header of the project, `debug_macros.h`, this header shall be included when using this macro.

```
—— debug_macros.h ——
#ifdef DEBUG
#define DEBUG_PRINT(str) fputs(str, stderr)
#else
#define DEBUG_PRINT(str) ((void) 0) // no action
#endif // DEBUG
```

[Code example]

```
void func(void) {
  DEBUG_PRINT(">> func\n") ;
  …
  DEBUG_PRINT("<< func\n") ;
}
```

(b) Using assert macro

In C++ language standard, assert macro is provided as a macro for program diagnosis. It is useful for making coding errors easier to detect during debugging. To facilitate debugging, define where to use the assert macro and follow this defined usage throughout the project. By doing so, it will be possible to collect consistent debug information during, such as, the integration test, and such information, as a result, will help make debugging easier.

The following is a brief explanation on how to use the assert macro, using a coding example that shows how this macro is used in a function definition written under the precondition that the null pointer is never passed as the argument.

```
void func(int *p) {
  assert(p != nullptr) ;
  *p = INIT_DATA ;
  …
}
```

If the NDEBUG macro is defined at compile time, the assert macro does nothing. On the other hand, if the NDEBUG macro is not defined and the expression passed to the assert macro is false, the program abends after outputting the file name and the line number of the source to the standard error. Note that the macro name is NDEBUG, and not DEBUG.

assert macro is a macro provided by the compiler in <assert.h>. By using the following example as a reference, examine how to abort the program and determine whether to use the macro provided by the compiler or to provide your own assert function.

```
#ifdef NDEBUG
#define assert(exp) ((void) 0)
#else
#define assert(exp) (void) ((exp)) || (_assert(#exp, __FILE__, __LINE__)))
#endif
void _assert(char *mes, char *fname, unsigned int lno) {
  fprintf(stderr, "Assert : %s : %s(%d)\n", mes, fname, lno) ;
  fflush(stderr) ;
  abort() ;
}
```

C++11 allows embedding static assert that can be evaluated by the compiler in the source code and confirming the offset of a structure member and the length of a string constant at the time the code is compiled.

```
static_assert( sizeof(t) <= 4, "The size of t is exceeding 4 bytes. " );
```

●Outputting logs after release

It is also useful to include descriptions for problem investigation in the release module that does not contain descriptions for debugging. One common method is to record the result of the investigation as log information. Log information is helpful for validation testing of the release module as well as for investigation of problems that occurred in the system provided to the customer.

In case of recording the log information, the following items should be determined in advance and defined as the coding convention.

●When to output logs

Logs should be output not only when an abnormal condition is detected, but also at the timing of, such as, data communication with an external system. The point is to output logs at appropriate timing (such as, when key events occur) that will make it easier to trace the history and faster to identify the root cause of the detected abnormality.

●What to output in logs

Information on the process executed immediately before the occurrence of the abnormal condition, the data values processed at that time, and information for tracing memory usage are some of the log information that should be recorded to enhance the traceability of the history and facilitate the investigation of the cause of the abnormality.

●Macro or function for outputting log information

Localize the log information output as a macro or a function. It is often preferable to make the log output destination changeable.

M5.1.2

(1) The # and ## preprocessor operators should not be used.
(2) A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
【MISRA C: 2012: R.20.11】

Preference guide	
Rule specification	Choose

Compliant example

```
Compliant example of (2)
#define AAA(a, b) a#b
#define BBB(x, y) x##y
```

Non-compliant example

```
Non-compliant example of (1), (2)
#define XXX(a, b, c) a#b##c
```

The order of evaluation of # operator and ## operator is not defined. Therefore, do not mix # and ## operators, nor use them more than once.

M5.1.3 Inline function shall be used rather than using function-like macro.

Preference guide	
Rule specification	

Compliant example

```
inline int func(int arg1, int arg2) {
  return arg1 + arg2;
}
```

Non-compliant example

```
#define func(arg1, arg2) (arg1 + arg2)
```

The process will become easier to trace, for example, if the process is stopped at the beginning of a function for debugging. Therefore, use the inline function rather than using function-like macro.
In addition, since the compiler will perform type check, coding errors will become easier to detect if inline function is used instead of function-like macro.
If there is a need to support multiple types, one way is to use the inline function template.

```
template<typename T>
inline T& func(const T& arg1, const T& arg2) {
  retrun arg1 + arg2 ;
}
```

[Related rule] E1.1.1

M5.2 Be careful when using dynamic memory allocations.

M5.2.1

(1) Dynamic memory shall not be used.

(2) If dynamic memory is used, 《the maximum amount of memory that can be used, process to be taken when running out of memory, and debugging procedure shall be defined》.

Preference guide	
Rule specification	Choose Define

Using dynamic memory involves the risk of accessing invalid memory as well as the risk of memory leak that leads to depletion of system resources, which may be caused by forgetting to return the obtained memory space to the system.

●Example of rule definition:

<Link to the sample or reference document>

●Do not allocate the object used only in the function as new.

<Non-compliant example>

```
func() {
  CLS *p = new CLS();  // Non-compliant
  …
  delete p ;
}
```

<Compliant example>

```
func() {
  CLS clobj ;  // Compliant
  …
}
```

●Reference: Problems when using dynamic memory

Described below are problems that tend to occur when dynamic memory is used.

●Buffer overflow

This occurs as a result of referencing or updating areas beyond the range of obtained memory. In particular, when an area outside the range is accidentally updated, this failure does not occur at the time of update but will occur when the memory destroyed by the update is referenced. The problem with dynamic memory is that it is very difficult to locate which part of the memory has been destroyed.

●Forgetting to initialize

When a class area is allocated with new and an appropriate constructor is defined, there is no risk of initialization being forgotten. However, be careful when an area other than class is allocated with new, because initialization will not be performed automatically.

●Memory leak

There is a risk of this problem being caused by forgetting to return the obtained memory space. This problem does not occur with programs that terminate each time after running once. However, with programs that keep running, memory leak can occur and become the cause of memory depletion and system malfunction.

●Use after return

When the obtained memory space is returned due to, such as, delete, the returned memory space may be reused later, such as, when new is called. In case the deleted memory address is used to update the memory, the memory space will be destroyed if it is already being reused for other purpose. As explained in the case with buffer overflow, it is very difficult to locate which part of the memory has been destroyed.

The code that leads to these problems does not cause a compile error. In addition, problems do not occur at the location where the bugs were implanted, making them undetectable in tests that are just for checking the normal specifications. They cannot be discovered unless the code is carefully reviewed or tests are performed after inserting a test code specifically written to detect such problems or after adding a special library to the program for similar purpose.

[Related rules] M4.8.2 M4.8.3

Portability

One of the distinctive aspects of embedded software is that there are diverse options in the platform used for software operation. This also means that there are many possible combinations of MPU options and OS options to select the hardware and software platforms from. As the number of functionalities realized by the embedded software increases, opportunities to port the existing software to other platforms by modifying or remodeling it to make it compatible with multiple platforms are also on the rise.
Due to this trend, software portability is becoming an extremely important element also at the source code level. In particular, writing in a style that is implementation-dependent is one of the most common mistakes made on a regular basis.

● Portability P1 : Write in a style that is not dependent on the compiler.
● Portability P2 : Localize the code that has a problem with portability.

Portability P1 – Write in a style that is not dependent on the compiler.

Use of compilers is unavoidable when programming in C++ language. Various compilers are available in the world and each has its own features. If the source code is written poorly, the code may become dependent on the features of the compiler used, and may cause unexpected results when a different compiler is used.
For this reason, programming must be performed carefully with an awareness that the code must be written in a style that is not implementation-dependent.

Portability P1.1	Do not use functionalities that are advanced features or implementation-defined.
Portability P1.2	Use only the characters and escape sequences defined in the language standard.
Portability P1.3	Confirm and document data type representations, behavioral specifications of advanced functionalities and implementation-dependent parts.
Portability P1.4	For source file inclusion, confirm the implementation-dependent parts and write in a style that is not implementation-dependent.
Portability P1.5	Write in a style that does not depend on the environment used for compiling.
Portability P1.6	Do not use unrecommended functionalities.

P1.1 Do not use functionalities that are advanced features or implementation-defined.

P1.1.1

(1) Functionalities not specified in the language standard shall not be used.
(2) If functionalities not specified in the language standard are used, 《the functionalities used and their usage shall be documented.》.

Preference guide	
Rule specification	Choose Docu-ment

At present, C++11 is the C++ language standard that is widely used. But there are still many compilers that also support the older version, C++03, including some compilers that support both C++11 and C++03, and allow the users to choose which version to use for processing.
Another realistic approach would be to choose rule (2) and allow the use of functionalities defined in the latest language standard, C++14, that are specifically supported by the compiler used.
Regarding the acceptable ways of using the functionalities that are not specified in the language standard, the details are provided in the following related rules.

[Related rules] P1.1.3 P1.2.1 P1.2.2 P1.3.2 P2.1.1 P2.1.2

P1.1.2

All usage of implementation-defined behavior shall be documented.

Preference guide	○
Rule specification	Docu-ment

In the language standard, there are implementation-defined items that vary depending on the compiler. Below are some examples of implementation-defined items and the information that must be documented when they are going to be used.
● How to represent floating-point numbers
● For C++03, how to handle signs of remainders for integer division.
It is recommended to round in the direction of zero.
● Search order of files for the #include directive
● `#pragma`

P1.1.3 To use a program written in another language, 《its interface shall be documented and its usage shall be defined》.

Preference guide	
Rule specification	Define / Docu-ment

The language standard does not define the interface for using programs written in languages other than C++ or C from a program written in C++. In other words, using a program written in another language requires the use of an advanced functionality, which means that portability will be impaired. Therefore, when using such a program, document the specifications of the compiler and define its usage, regardless of the possibility of porting.
For using the functions and variables described in C++ language from a program written in C, describe the linkage specification (`extern ″C″`) when defining them.

[Related rules] P1.1.1 P2.1.1

P1.2 Use only the characters and escape sequences defined in the language standard.

P1.2.1 To use characters other than those defined in the language standard for writing a program, the compiler specifications shall be confirmed and 《their usage shall be defined》.

Preference guide	
Rule specification	Define / Docu-ment

Compliant example

```
string index[10];  // table of nemes
int count = 0;
// Compliant
```

Non-compliant example

```
string index[10];  //  Name table.
int count = 0;
// Non-compliant:In Shift_JIS code, second
// byte of the character "表" is '\' in ASCII
// code, and may be interpreted as the line
// concatenation of // comment.
```

The basic character set defined in the language standard as usable for source code are upper and lower case letters of the Latin alphabet, decimal digits, graphic characters (`_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & ¦ = , ″ ′`), space character, and control characters that represent the horizontal tab, vertical tab, form feed and new line.
International characters and multibyte characters (like Japanese) can be used as identifiers and characters, but they may not be supported by some compilers. Therefore, if these characters are going to be used, check beforehand that they can be used in the following locations and define their usage.

·Identifiers

·String literals

- Processing when \ exists in the character codes of the string (whether special care is required or whether options are required at compile time, etc.)
- The necessity to write using wide string literals (with the prefix `L`, such as `L"string"`)
- The necessity to write using UTF-16 string literals (with the prefix `u`, such as `u"string"`)
- The necessity to write using UTF-32 string literals (with the prefix `U`, such as `U"string"`)

·Character constants

- The bit length of the character constant
- Processing when \ exists in the character codes of the character constant (whether special care is required or whether options are required at compile time, etc.)
- The necessity to write using wide character constants (with the prefix `L`, such as `L'あ'`).
- The necessity to write using `char16_t` character constants (with the prefix `u`, such as `u'あ'`).
- The necessity to write using `char32_t` character constants (with the prefix `U`, such as `U'あ'`).

·The file name of #include

For example, define the following rules.
・As the identifier, only the alphanumeric characters and underscore should be used.

In recent years, many processing systems have come to support multi-byte character code including Japanese with Unicode. In Japan, Shift_JIS has been widely used to express Japanese characters. But recently, projects adopting UTF-8 are increasing, such as in open system development projects.

[Related rules] M4.3.1 M4.3.2 P1.1.1

P1.2.2 Only escape sequences defined in the language standard shall be used.

Preference guide	
Rule specification	

Compliant example

```
char c = '\t';    // Compliant:
```

Non-compliant example

```
char c = '\e';  // Non-compliant：The escape is
                // sequence not defined in the
                // language standard. It is not
                // portable
```

The language standard defines the following seven nongraphic characters as escape sequences:

`\a` (alert)   `\b` (backspace)   `\f` (form feed)   `\n` (new line)
`\r` (carriage return)   `\t` (horizontal tab)   `\v` (vertical tab)

[Related rule] P1.1.1

P1.3 Confirm and document data type representations, behavioral specifications of advanced functionalities and implementation-dependent parts.

P1.3.1 Simple char type (that does not specify the signedness) shall be used only for storing character values. If a process that depends on signedness (implementation-defined) is required, unsigned char or signed char that specifies its signedness shall be used.

Preference guide	○
Rule specification	

Compliant example

```
char c = 'a';  // Used to store characters.
int8_t i8 = -1;  // To use it as an 8-bit
                 // data, use a type defined,
                 // for example, with typedef.
```

Non-compliant example

```
char c = -1;
if (c > 0) {
// Non-compliant：char can be signed or
// unsigned depending on the compiler, and
// this difference affects the result of the
// comparison.
```

Unlike other integer types like int, char that does not specify its signedness will be either signed or unsigned depending on the compiler (int type is the same as signed int type.) Therefore, using char that depends on the signedness is not portable. This is because char that does not specify its signedness is an independent type provided for storing characters (comprised of three types: char, unsigned char and signed char) and the language standard assumes such usage. For using char as a small integer type, such as, when a process that depends on signedness is required, use either unsigned char or signed char that specifies its signedness. In this case, it is desirable to use typedef as the type to localize the range of modification during porting.

A problem similar to this rule exists with the returned type of the standard function getc that is int and must not be received by char. However, this is rather a problem pertaining to function interface (assignment that may cause information loss).

【Reference materials for those wanting to know more in detail about this rule】

- MISRA C:2012 R10.1

[Related rule] P2.1.3

P1.3.2 The members of an enumeration (enum) type shall be defined with values that can be represented as int type.

Preference guide	
Rule specification	

Compliant example

```
enum T {
  E = INT_MAX  // Compliant:：Can be
               // represented with int.
};
```

Non-compliant example

```
enum T {
  E = LONG_MAX  // Non-compliant：Cannot be
                // represented with int.

};
```

In C++ language standard, the type of the members of an enumeration type is the type that can represent all the member values. For example, if all the members of an enumeration type can be represented as long type but not as int type, the type of all these members is long.
However, in C language standard, the members of an enumeration type must have values that can be represented as `int` type. Therefore, by considering the compatibility between C and C++ languages, limit the values of the members to the range that can be represented with `int` type.

[Related rules] P1.1.1 M2.2.4

P1.3.3

(1) Bit fields shall not be used.
(2) Bit fields shall not be used for data whose bit positions are meaningful.
(3) 《If it is being relied upon, the implementation-defined behavior and packing of bit fields shall be documented.》

Preference guide	○
Rule specification	Choose Document

Compliant example

```
Compliant example of (2)
struct S {
  unsigned int bit1 : 1;
  unsigned int bit2 : 1;
};
extern struct S * p;
  // Compliant if p points to a data that
  // is, for example, just a set of flags and
  // bit1 can be any bit in that data.

p->bit1 = 1;
```

Non-compliant example

```
Non-compliant example of (2)
struct S {
  unsigned int bit1 : 1;
  unsigned int bit2 : 1;
};
extern struct S * p;
  // If the bit positions are meaningful, for
  // example, when p points at IO ports; in
  // other words, if there is a meaning for
  // bit1 to point at either the lowest bit
  // or the highest bit of the data, the
  // program is non-portable.

p->bit1 = 1;  // As to which bit of
              // the data, p will point at,
              // is implementation-dependent.
```

The following behaviors of bit field vary depending on the compiler used:

1) Whether the bit field of an enum type or an int type that does not specify its signedness will be handled as signed;
2) Assignment order of the bit fields within a unit;
3) Boundary of a bit field in a storage unit.

If bit fields are used to access data whose bit positions are meaningful, such as, the IO ports, portability problem arises due to rules (2) and (3). Therefore, in such cases, do not use bit fields, but instead, use bitwise operations, such as, & and |.

[Related rule] R2.6.1

P1.4 For source file inclusion, confirm the implementation-dependent parts and write in a style that is not implementation-dependent.

P1.4.1 The `#include` directive shall be followed by either a *<filename>* or *"filename"* sequence.

Preference guide	●
Rule specification	

Compliant example

```
#include <stdio>
#include "myheade.h"
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h"
#endif
#include INCFILE
```

Non-compliant example

```
#include stdio   // Neither <> nor "" is
                 // placed.
#include "myheader.h" 1  // 1 is specified at
                         // the end.
```

In the language standard, the behavior is not defined for cases where the format of the header name does not match with neither of the two styles (< > or " ") after macro-expansion of the `#include` directive. Most compilers will output an error if it cannot match the format with neither of the two styles, while some others may not handle it as an error. Therefore, write the header name format in either of the two styles to ensure safety.

P1.4.2 《The usage of < > format and " " format for #include file specification shall be defined.》

Preference guide	
Rule specification	Define

Compliant example

```
#include <stdio>
#include "myheader.h"
```

Non-compliant example

```
#include "stdio"
#include <myheader.h>
```

There are two ways of writing `#include`. To unify the writing style, define rules, for example, that include the following:

- Specify the header provided by the compiler by enclosing it with < >;
- Specify the header created in the project by enclosing it with ″ ″;
- Specify the header provided by the purchased software by enclosing it with ″ ″.

P1.4.3 Characters ′, \, ″, /*, // and : shall not be used for file specification in #include.

Preference guide	○
Rule specification	

Compliant example

```
#include "inc/my_header.h"   // Compliant:
```

Non-compliant example

```
#include "inc\my_header.h"   // Non-compliant
```

The language standard does not define the behavior when the characters mentioned above are used (more specifically, in the following cases); that is to say, the operation result is not certain when these characters are used in the following cases, which consequently make the code non-portable:

- When characters ′ , \, ″, /* or // appear in the string enclosed with < > ;
- When characters ′ , \, ″, /* or // appear in the string enclosed with ″ ″.

Also, the behavior of the character : (colon) differs depending on the compiler, and makes the code non-portable.

P1.5 Write in a style that does not depend on the environment used for compiling.

P1.5.1 The absolute path shall not be written for #include file specification.

Preference guide	
Rule specification	

Compliant example

```
#include "h1.h"
```

Non-compliant example

```
#include "/project1/module1/h1.h"
```

If an absolute path is written in the code, there will be a need to modify the path when the program is compiled after changing the directories.

P1.6 Do not use unrecommended functionalities.

P1.6.1 Deprecated functionalities shall not be used.

Preference guide	○
Rule specification	

Compliant example

```
// Example 1: increment operator in type bool
bool  x = false;
x = true;  // Compliant

// Example 2: register keyword
int y; // Compliant
```

Non-compliant example

```
// Example 1: increment operator in type bool
bool  x = false;
x++;  // Non-compliant

// Example 2: register keyword
register int y; // Non-compliant: register
                // keyword is deprecated.
```

There is no guarantee as to when the deprecated functionalities will continue to be supported till. It is therefore safer not to use them.
Deprecrated functionalities are listed in Annex D of the C++ Standard. For example, in C++11, increment operator in type bool and register keyword are deprecated.

Portability P2 Localize the code that has a problem with portability.

The principle is not to write implementation-dependent source code as much as possible. But in some cases, writing implementation-dependent source code is unavoidable. A typical example is when an assembly language program is called. In such case, it is recommended to localize the implementation-dependent parts of the code as much as possible.

Portability P2.1 Localize the code that has a problem with portability.

P2.1 Localize the code that has a problem with portability.

P2.1.1

When assembly language programs are called from C++ language, 《how to localize such parts shall be defined》, such as, by expressing them as functions of C++ language that contain only inline assembly language code or by describing them using macros.

Preference guide	○
Rule specification	Define

Compliant example

```
#define SET_PORT1 asm("    st.b 1, port1")
void f() {
  …
  SET_PORT1;
  …
}
```

Non-compliant example

```
void f() {
  …
  asm("    st.b 1,port1");
  …
}
// asm and other processes are mixed.
```

Many compilers provide extended support to support `asm(`*string*`)` format as a method to include the assembly language code. However, there are some compilers that do not provide such support. Moreover, the same format may lead to different behavior depending on the compiler used. As a result, the code becomes non-portable.

[Related rules] P1.1.1　P1.1.3

P2.1.2

Keywords extended by the compiler shall be used by localizing them after 《defining the macros.》

Preference guide	○
Rule specification	Define

Compliant example

```
// interrupt is defined as a keyword
// extended by a specific compiler.
#define INTERRUPT interrupt
INTERRUPT void int_handler(void) {
  …
}
```

Non-compliant example

```
// interrupt is defined as a keyword extended
// by a specific compiler.
// It is used without being defined as a macro.
interrupt void int_handler(void)  {
  …
}
```

Some compilers provide extended keywords instead of using the `#pragma` directive. But the code will become non-portable when these keywords are used. Therefore, when using them, localize them, such as, by defining them as macros. For the macro name, use the keyword written in uppercase letters, as shown above in the compliant example.

[Related rule] P1.1.1

P2.1.3

(1) The basic types (char, int, long, long long, float, and double) shall not be used. Instead, the types defined by typedef shall be used. 《The types defined by typedef that are used in the project shall be defined.》

(2) When using any of the basic types (char, int, long, long long, float, and double) in a form that is dependent on its size, the type defined by typedef for each of these basic types shall be used. 《The types defined by typedef that are used in the project shall be defined.》

Preference guide	○
Rule specification	Choose Define

Compliant example

```
Compliant example of (1), (2)
uint32_t flag32;  // When assuming that the
                  // size is 32 bits,
                  // use uint32_t.

Compliant example of (2)
int i; → int i;
for (i = 0; i < 10; i++) {
// i is used as the index.
// Any basic type defined in the language
// specification can be used, because i can
// be represented by 8 bits, 16 bits or 32
// bits.
```

Non-compliant example

```
Non-compliant example of (1), (2)
unsigned int flag32;  // Use int, assuming
                      // that the size is 32
                      // bits.
```

The size and internal representation of integer types and floating point types vary depending on the compiler.
C++11 specifies the following typedefs to be provided as the language standard. Therefore, these type definitions should be used.

```
int8_t    int16_t    int32_t    int64_t    uint8_t    uint16_t    uint32_t    uint64_t
```

When using a compiler that supports C++03, it is advisable to refer to them as the `typedef` names for these basic types.

[Related rule] P1.3.1

Efficiency

Embedded software is characteristic for being embedded in a product and operating together with hardware to serve its purposes in the real world. The increasing demand for further product cost reduction has imposed various restrictions, not only on, such as, MPU or memory, but also on software.
In addition, requirements, such as, on real-time property have placed stricter time constraints that need to be met. Embedded software must therefore be coded with particular attention on resource efficiency like efficient use of memory and time efficiency that takes account of time performance.

● Efficiency E1 : Write in a style that takes account of resource and time efficiencies.

Efficiency E1 Write in a style that takes account of resource and time efficiencies.

Depending on how the source code is written, the object size may increase and the execution speed may slow down. If there are restrictions on memory size and processing time, the code must be written thoughtfully with additional considerations given to these restrictions.

Efficiency E1.1 Write in a style that takes account of resource and time efficiencies.

E1.1 Write in a style that takes account of resource and time efficiencies.

E1.1.1 Macro functions shall be used only in parts related to speed performance.

Preference guide	●
Rule specification	

Compliant example

```
extern void func1(int,int);  // Function.
#define func2(arg1, arg2)  // Function macro.
func1(arg1, arg2);

for (i = 0; i < 10000; i++) {
  func2(arg1, arg2);  // Speed performance
                      // is critical for
                      // this process.
}
```

Non-compliant example

```
#define func1(arg1, arg2)  // Function macro.
extern void func2(int, int);  // Function.
func1(arg1, arg2);

for (i = 0; i < 10000; i++) {
  func2(arg1, arg2);  // Speed performance
                      // is critical for
                      // this process.
}
```

Function is safer than macro function. So, use function as much as possible (see M5.1.3). However, processing of function calls and returns may slow down the speed performance. Therefore, if speed performance is an issue that has to be improved, use macro function instead.
Note, however, that when too many macro functions are used, the object size may increase because the expanded code will be spread to wherever the macro function is used.

[Related rule] M5.1.3

E1.1.2 Operations that remain unchanged shall not be performed within an iterated process.

Preference guide	●
Rule specification	

Compliant example

```
var1 = func();
for (i = 0; (i + var1) < MAX; i++) {
  …
}
```

Non-compliant example

```
// Function func returns the same result.
for (i = 0; (i + func()) < MAX; i++) {
  …
}
```

Repeating the same process that returns the same result is inefficient. Although optimization of the compiler is often reliable for preventing such inefficiency, attention is still necessary in some cases, like in the non-compliant example shown above, where the compiler used cannot determine the invariance.

E1.1.3 If the function argument is type class, pass by reference or pass by pointer shall be used instead of pass by value.

Preference guide	●
Rule specification	

Compliant example

```
class CLS { … } ;
  int func1(const CLS &c);
  // Compliant : Pass by reference.
  int func2(const CLS *p);
  // Compliant : Pass by pointer.
class CLS obj;
…
  func1(obj);
  // Compliant : Pass by reference.
  func1(&obj);
  // Compliant :Pass by pointer.
```

Non-compliant example

```
class CLS { … } ;
  int func(const CLS c);
  // Non-compliant :Pass the class as it is.
class CLS obj;
  …
  func(obj);
  // Non-compliant : Pass the class as it is.
```

It is generally desirable to use pass by reference for function argument. But when null pointer is used as the interface, use pass by pointer. For referencing only what is called, use const qualifier. When pass by reference is used, the creation of temporary objects may lead to significant overhead. In C++11, rvalue reference has been introduced as a new feature to reduce the copy overhead of temporary objects.

【Reference materials for those wanting to know more in detail about this rule】

● Effective Modern C++ Chapter 5

[Related rule] R3.7.1

E1.1.4 《The policy of selecting either `switch` or `if` statement shall be determined and defined by taking readability and efficiency into consideration.》

Preference guide	
Rule specification	Define

switch statements often provide higher readability than if statements. In addition, recent compilers tend to output optimized code using, such as, table jump or binary search when they process switch statements. Take these matters into consideration when defining this rule.

Example of the rule: `switch` statement shall be used instead of if statement when:

- a process branches according to the value of the expression (integer value), and
- the number of branches is three or more.

However, this rule shall not apply if:

- using the `switch` statement causes an efficiency issue that impedes the improvement of program performance.

E1.1.5 Variable definition and function definition shall not be written in the header file.

Preference guide	○
Rule specification	

Compliant example

```
Compliant example(1)
--- file1.h ---
extern int x;  // Variable declaration.
int func(void) ;  // Function declaration.

--- file1.c ---
#include "file1.h"
int x;  // Variable definition.
int func(void) { // Function definition.
  …
}

Compliant example(2)
-- a.h --
namespace NS1 {  // Compliant: Namespace
                //  with a name is used.
  int x;  // x has only one instance.
  void func() { … }  // func has only one
                     // instance.
};
-- a.cpp --
#include "a.h"
void f1() {
  NS1::x = 10;  // A value is set to x of NS1
  NS1::func();  // func() of NS1 is called.
  …
}
-- b.cpp --
#include "a.h"
void f2() {
  NS1::x = 20;  // A value is set to x of NS1
  NS1::func();  // func() of NS1 is called.
  …
}
```

Non-compliant example

```
Non-compliant example(1)
--- file1.h ---
int x;  // External variable definition.
static int func(void) {  // Function definition.
  …
}

Non-compliant example(2)
The instances of x and func() are generated
in both files, a.cpp and b.cpp. What this
means is that the memory space for two x
will be allocated, and that, as a result,
the code size of function func() will be
doubled.

-- a.h --
namespace {
  // Non-compliant: Unnamed namespace is used.
  int x;
  void func() { … }
};
-- a.cpp --
#include "a.h"
void f1() {
  x = 10;
// A value is set to x defined in this file.
  func();
// func defined in this file is called.
  …
}
-- b.cpp --
#include "a.h"
void f2() {
  x = 20;
// A value is set to x defined in this file.
  func();
// func defined in this file is called.
  …
}
```

A header file may be included in multiple source files. Therefore, if variable declaration and function declaration are written in the header file, the size of the object code generated after compilation may become unnecessarily large and risky. In the header file, basically write only the declaration and type definition. If necessary, though, the definition of inline function and template function may also be written in there.

E1.1.6 For initialization of data members, initialization shall be written in member declaration or constructor initializer shall be used.

Preference guide	○
Rule specification	

Compliant example

```
class CLS1 {
public:
   CLS1() { ... }
   CLS1(const CLS1 &cls1) { ... }
   CLS1(int i) { ... }
   ...
};

class CLS2 {
public:
  CLS2 (const CLS1 & cls1) : cls1_1(cls1)
    { ... } ;
               // Compliant:
               // Initialization is
               // executed by
               // initialization list.
    ...
private:
  CLS1 cls1_1;
  CLS1 cls1_2 = CLS1(10);
               // Compliant:
               // Initialization is
               // written in member
               // declaration.
  ...
} ;
```

Non-compliant example

```
class CLS1 {
public:
   CLS1() { ... }
   CLS1(const CLS1 &cls1) { ... }
   CLS1(int i) { ... }
   ...
};

class CLS2 {
public:
  CLS2 (const CLS1 & cls1)    {
      cls1_1 = cls1;
      cls1_2 = CLS1(10);
 } ;
   // Non-compliant: Initialization is not
   // executed because initialization is
   // neither written in member declaration
   // nor in the constructor initializer.
   // To initialize cls1_1 and cls1_2
   // respectively, CLS1() is called, and
   // subsequently, the assignment written in
   // the constructor itself is executed.
    ...
private:
  CLS1 cls1_1;
  CLS1 cls1_2;
  ...
} ;
```

Data members of the type class that have not been initialized by the constructor initializer or by execution of initialization written in member declaration are implicitly initialized by default constructor at the beginning of constructor execution. Therefore, when these uninitialized members are initialized in the constructor by using the assignment operator, the aforesaid implicit initialization becomes redundant and unnecessary.

[Related rule] R1.4.1

E1.1.7 Overload function that corresponds with the type shall be defined to prevent implicit type conversion from occurring.

Preference guide	
Rule specification	

Compliant example

```
class CLS {
public:
  explicit CLS(int i) : cls_i(i) { … }
  CLS &operator += (const CLS &rhs) {
    return this += rhs.i;
  }
  // Compliant：Function that matches with
  // the parameter type is defined.to prevent
  // implicit type conversion from occurring.
  CLS &operator += (int i) {
    cls_i += i;
    return * this;
  }
  …
private:
  int cls_i;
};

void func() {
  …
  CLS cls_obj(10);
  cls_obj += 10;
  // Compliant：operator += (int) is called
  // and temporary object is not generated.
```

Non-compliant example

```
class CLS {
public:
  explicit CLS(int i) : cls_i(i) { … }
  CLS &operator += (const CLS &rhs) {
    cls_i += rhs.cls_i;
    return * this;
  }
  // Non-compliant：Function that matches
  // with the parameter type is not defined.
private:
  int cls_i;
};

void func() {
  …
  CLS cls_obj(10);
  cls_obj += 10;
    // Non-compliant：operator +=(CLS),a
    // temporary object with default value
    // 10 is generated.
```

When implicit type conversion occurs, unnecessary temporary object will be generated. Therefore, when overloading, avoid implicit type conversion by defining the function that matches with the parameter type. As to the extent of parameter types that should be covered respectively in function definitions for the purpose of avoiding implicit type conversion, determine it based on the right balance between the desired level of improvement in efficiency and the size and maintainability of the written code.

【Reference material for those wanting to know more in detail about this rule】

● More Effective C++ Item 21

[Related rules] M1.8.7 M1.8.8

E1.1.8 Prefix form shall be used for increment and decrement operators.

Preference guide	○
Rule specification	

Compliant example

```
class CLS {
public:
  CLS &operator ++() {  // Prefix form.
    … // Increment operation is executed.
    return *this;
  }
  CLS operator ++(int) {   // Postfix form.
    CLS old_value(*this);  // The value just
                           // before it
                           // was updated is
                           // retained.
    ++ (*this);
    return old_value;  // The value just
                       // before it was
                       // updated is returned.
  }
  …
}

void func() {
  for (int i = 0; i<MAX; i++) {
    ++ cls;  // Compliant : Prefix ++ .
// Some kind of processing is executed
// after ++ operation.
```

Non-compliant example

```
class CLS {
public:
  CLS &operator ++() {  // Prefix form.
    … // Increment operation is executed.
    return *this;
  }
  CLS operator ++(int) {   // Postfix form.
    CLS old_value(*this);  // The value just
                           // before it was
                           // updated is
                           // retained.
    ++ (*this);
    return old_value;      // The value just
                           // before it was
                           // updated is
                           // returned.
  }
  …
}

void func() {
  for (int i = 0; i<MAX; i++) {
    cls++;  // Non-compliant : Unnecessary
            // postfix ++ Some kind of
            // processing is executed after
            // ++ operation.
```

When the increment and decrement operators are written in postfix form, the original value is returned after once retaining it in a temporary object. If there is no need to use the original value, write the increment and decrement operators in prefix form.
This rule, however, does not prohibit the use of postfix form for types like int and pointer defined in the language that have been customarily written in postfix form.

Example:

```
for (int i = 0 ; i<MAX ; i++) // Compliant:  Customary code written for int type.
```

E1.1.9 The use of virtual inheritance shall be allowed only when the classes that have the same base class are multiply inherited.

Preference guide	
Rule specification	

Compliant example

```
class Base { … };
class D1 : public virtual Base { … };
class D2 : public virtual Base { … };
class DD : public D1, D2 { … };
    // Compliant : Virtually inherited from
    // the same base class.
class Base1 { … };
class Base2 : public Base1 { … };
class CLS : public Base2 { … };
    // Compliant : Virtual inheritance is not
    // used.
```

Non-compliant example

```
class Base { … };
class D1 : public virtual Base { … };
class D2 : public virtual Base { … };
// D1 and D2 will be non-compliant if a class
// (DD in the compliant example) that is
// required to organize the data members of
// a base class collectively as a single class
// member is not defined.

class Base1 { … };
class Base2 : public virtual Base1 { … };
class CLS : public virtual Base2 { … };
    // Non-compliant : Virtual inheritance is
    // used when the  inheritance is not a
    // multiple inheritance.
```

Virtual inheritance increases the object size and slows down the execution speed. Therefore, do not use it unless the classes that have the same base class are multiply inherited and the data members of the same bass class are viewed as they belong to a single common bass class.

【Reference material for those wanting to know more in detail about this rule】

- More Effective C++ Item 24

[Related rule] M3.5.1

E1.1.10 Any code that is unrelated to the template parameter shall not be written in the template definition.

Preference guide	○
Rule specification	

Compliant example

```
templae <class T>
class CLS_T {
public:
  CLS_T(const T &tobj) : cls_t_obj(tobj) {  }
  T get_value() {return cls_t_obj;}
private:
  T cls_t_obj;
};

  // Compliant：util_func is unrelated to
  // the template parameter and is defined
  //outside of the template.
void util func() {
  // Processing code that is unrelated to
  // the template parameter.
  …
}
```

Non-compliant example

```
templae <class T>
class CLS_T {
public:
  CLS_T(const T &tobj) : cls_t_obj(tobj) {  }
  T get_value() {return cls_t_obj;}
  // Non-compliant：util_func is not related
  // to template parameter.

  void util func() {
  // Processing code that is unrelated to the
  // template parameter.
    …
  }
private:
  T cls_t_obj;
};
```

Writing any code unrelated to the template parameter within the template may make the expanded code oversized.
Therefore, any code unrelated to the template parameter should be written outside of the template.

Appendix List of practices and rules

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
[Reliability R1] Initialize areas and use them by taking their sizes into consideration.					
R1.1 Use areas after initializing them.	R1.1.1	Automatic variables shall be initialized at the time of declaration, or the initial values shall be assigned just before using them.	9.1	R9.1	8-5-1
	R1.1.2 (C edition)	const variables shall be initialized at the time of declaration.			
R1.2 Describe initializations without excess or deficiency.	R1.2.1 (C edition)	Arrays with specified number of elements shall be initialized with values that match the number of the elements.			
	R1.2.2	Initialization of enumeration type (enum type) members shall be by either: not specifying any constants; specifying all the constants; or specifying only the first member.	9.3	R8.12	8-5-3
R1.3 Pay attention to the range of the area pointed by a pointer.	R1.3.1	(1) Integer addition to or subtraction from (including ++ and --) pointers shall not be made; Array format with [ ] shall be used for references and assignments to the allocated area. (2) Integer addition to or subtraction from (including ++ and --) pointers shall be made only when the pointer points to the array and the result must be pointing within the range of the array.	17.1 17.4	R18.1 R18.4	5-0-15 5-0-16
	R1.3.2	Subtraction between pointers shall be used only when both pointers are pointing at elements in the same array.	17.2	R18.2	5-0-17
	R1.3.3	Comparing which pointer is greater or less than the other pointer shall be used only when two pointers are both pointing at either the elements in the same array, the members with the same access control defined in the same structure or class, or the members of the same structure.	17.3	R18.3	5-0-18
R1.4 Use the object after constructing it completely.	R1.4.1	All the data members of a class shall be initialized in its constructor. The initialization procedure shall be as follows: 1. Initialization shall be written in the declaration of members that are always initialized with the same value. The constructor initializer shall be used to perform the initialization of other members. However, this shall not apply in case of initializing multiple non-class type members with the same value. 2. The constructor initializer shall have the base class and data members written in the order they are declared. 3. The constructor initializer shall not use any other data members of the same class for initialization. Or, if there is a need to use other any of these data members, they shall be limited to only those declared before the specific data member targeted for initialization.			8-5-1
	R1.4.2	Non-static data members shall be all copied whenever a copy constructor or copy assignment operator is used.			
	R1.4.3	Member function for only reading data members shall be called after the constructor initializes the object completely and before the destructor starts destroying the object.			
	R1.4.4	Virtual function shall not be called from the constructor and destructor.			12-1-1
	R1.4.5	The constructor and copy assignment operator shall respond to unsuccessful object construction.			15-1-1 15-3-1
	R1.4.6	Catch handler described in the constructor or destructor shall not reference data members of that class.			15-3-3
R1.5 Pay attention to object creation and destruction.	R1.5.1	The same form (whether with or without [ ]) shall be used for new and corresponding delete .			
[Reliability R2] Use data by taking their ranges, sizes and internal representations into consideration.					
R2.1 Make comparisons that do not depend on internal representations.	R2.1.1	Floating-point expressions shall not be used to perform equality or inequality comparisons.	13.3	D1.1	6-2-2
	R2.1.2	Floating-point variable shall not be used as a loop counter.	13.4	R14.1	6-5-1
	R2.1.3	memcmp shall not be used to compare class-type objects.			
R2.2 When values such as logical values are defined as a range, do not make a judgment by finding whether or not a value is equivalent to any particular value (representative value) within this range	R2.2.1 (C edition)	Comparison with a value defined as true shall not be made in expressions that examine true or false.			
R2.3 Use the same data type to perform operations or comparisons.	R2.3.1	Unsigned integer constant expressions shall be described within the range that can be represented with the result type.	12.11	R12.4	5-19-1
	R2.3.2	When using conditional operator (? : operator), the logical expression shall be enclosed in parentheses( ) and both return values shall be the same type.			
	R2.3.3	Loop counters and variables used for comparison of loop iteration conditions shall be the same type.			

To indicate the relationship with MISRA rules, the rule number of corresponding (the same or alike) MISRA rule is used (for both MISRA C rules and MISRA C++ rules, respectively).
When "(C edition)" is written in the space for indicating the related rule numbers, it means that the rule is included in ESCR C Language edition but has been deleted from ESCR C++ Language edition. When "(vacant)" is written in the space for indicating the related rule numbers, it means that the rule has been deleted from Ver. 1.0 of both ESCR C and ESCR C++ Language editions.

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
R2.4 Describe code by taking operation precision into consideration.	R2.4.1	When the type of an operation and the type of the destination to which the operation result is assigned (assignment destination) are different, the operation shall be performed after casting them to the type of expected operation precision.	10.3 10.4	R10.8	5-0-7 5-0-8
	R2.4.2	When performing arithmetic operations or comparisons of expressions mixed with signed and unsigned, an explicit cast to the expected type shall be performed.			
R2.5 Do not use operations that have the risk of information loss.	R2.5.1	When performing assignments (=operation, actual arguments passing of function calls, function return) or operations to data types that may cause information loss, they shall be first confirmed that there are no problems, and a cast shall be described to explicitly state that they are problem-free.	10.1 10.2	R10.3 R10.4 R10.6 R10.7	5-0-3
	R2.5.2	Unary operator '-' shall not be used in unsigned expressions.	12.9	R10.1	5-3-2
	R2.5.3	When one's complement (~) or left shift (<<) is applied to unsigned char or unsigned short type data, an explicit cast to the type of the operation result shall be performed.	10.5		5-0-10
	R2.5.4	The right-hand side of a shift operator shall be zero or more, and less than the bit width of the left-hand side.	12.8	R12.7	5-8-1
R2.6 Use types that can represent the target data.	R2.6.1	(1) The types used for bit field shall only be signed int or unsigned int. If a bit field of 1 bit width is required, unsigned int type shall be used, and not the signed int type. (2) bool type, integer type that specified the signedness (signed or unsigned) or signedness-specified enum type shall be used for bit field. If a bit field of 1 bit width is required, the integer type that specified unsigned or bool type shall be specified. (3) Integer type that specified the signedness (signed or unsigned), bool type or enum type shall be used for bit field. If a bit field of 1 bit width is required, the integer type that specified unsigned or bool type shall be specified.	6.4 6.5	R6.1 R6.2	9-6-2 9-6-3 9-6-4
	R2.6.2	Data used as bit sequences shall be defined with unsigned type, and not with the signed type.	12.7	R10.1	5-0-21
R2.7 Pay attention to pointer types.	R2.7.1	(1) Pointer type shall not be converted to other pointer type or to integer type, and vice versa, with the exception of the following cases: ・Conversion from pointer to data type to void * type; ・Conversion between pointers to class typewith base-derived relationship. (2) Pointer type shall not be converted to other pointer type or to integer type with less data width than that of the pointer type, with the exception of the following cases: ・Mutual conversion between void * types in pointer to data type; ・Conversion between pointers to class typewith base-derived relationship. (3) Pointer to data type can be converted to pointer to other data type or to void * type, but pointer to function type shall not be converted to pointer to other function type or to pointer to data type. In case of converting pointer type to integer type, such conversion shall not be performed if the data width of the integer type is less than that of the pointer type.	11.1 11.2 11.3 11.4	R11.1 R11.2 R11.3 R11.4 R11.5 R11.6 R11.7	5-2-6 5-2-7 5-2-8 5-2-9
	R2.7.2	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	11.5	R11.8	5-2-5
	R2.7.3	Comparison to check whether a pointer is negative or not shall not be performed.			
	R2.7.4	(1) A pointer to a derived class may be converted to a pointer to its base class. However, a pointer to the base class may not be converted to a pointer to its derived class. (2) A pointer to a derived class may be converted to a pointer to its base class. Moreover, if dynamic _ cast operator is used, a pointer to a base class may also be converted to a pointer to its derived class.			5-2-2 5-2-3
R2.8 Write in a way that will enable the compiler to check that there are no conflicting declarations, usages and definitions.	R2.8.1 (C edition)	Functions with no parameters shall be declared with a void type parameter.			
	R2.8.2	(1) Functions shall not be defined with a variable number of arguments. (2) When using functions with a variable number of arguments, 《they shall be used after documenting the intended behaviors based on the compiler used》	16.1	R16.1	8-4-1
	R2.8.3	One prototype declaration shall be made at one location, so that it will be referenced by both its function calls and function definition.	8.1 8.3 8.4	R8.2 R8.3 R8.4 R17.3	3-2-1 3-2-2 3-3-1 3-9-1

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
[Reliability R3] Write in a way that ensures intended behavior.					
R3.1 Write in a way that is conscious of area size.	R3.1.1	(1) In an extern declaration of an array, the number of elements shall always be specified. (2) In an extern declaration of an array, the number of elements shall always be specified, except for extern declarations of arrays that correspond to the array definition that includes initialization and has omitted the number of elements.	8.12	R8.11	3-1-3
	R3.1.2	For a loop to sequentially access array elements, its iteration conditions shall include the judgment on whether the access is within the range of the array or not. However, for a loop to sequentially access array elements from the beginning of the array, range-based for loop shall be used.			
	R3.1.3 (C edition)	The size of the array initialized with a designated initializer shall be clearly indicated.			
	R3.1.4 (C edition)	Variable length array type shall not be used.		R18.8	
R3.2 Prevent operations that may cause runtime error from falling into error cases.	R3.2.1	Operations shall be performed after confirming that the right-hand side expression of division or remainder operation is not 0.	21.1	D4.1	0-3-1
	R3.2.2	Destination pointed by a pointer shall be referenced after checking that the pointer is not the null pointer.	21.1	D4.1	0-3-1
R3.3 Check the interface restrictions when a function is called.	R3.3.1	If a function returns error information, then that error information shall be tested.	16.1	R17.1	0-3-2
	R3.3.2	The restrictions of the parameters shall be checked before starting the function processing.	20.3	D4.11	0-3-1
R3.4 Do not perform recursive calls.	R3.4.1	Functions shall not call themselves, either directly or indirectly.	16.2	R17.2	7-5-4
R3.5 Pay attention to branch conditions and describe how to handle cases that do not follow the predefined conditions when they occur.	R3.5.1	The else clause shall be written at the end of an ifelse if statement. If it is known that the else condition does not normally occur, the description of the else clause shall be either one of the following: 《(i) Unexpected condition handling process shall be written in the else clause. (ii) A project-specific comment specified by the project shall be written in the else clause.》	14.10	R15.7	6-4-2
	R3.5.2	The default clause shall be written at the end of a switch statement. If it is known that the default condition does not normally occur, the description of the default clause shall be either one of the following: 《(i) Unexpected condition handling process shall be written in the default clause. (ii) A project-specific comment specified by the project shall be written in the default clause.》	15.3	"R16.4 R16.5"	6-4-6
	R3.5.3	Equality operators (== !=) shall not be used for comparison of loop counters.			6-5-2
R3.6 Pay attention to the order of evaluation.	R3.6.1	Variables whose values are changed shall not be referred to or modified in the same expression.	12.13	R13.3	5-2-10
	R3.6.2	Function calls with side effects and volatile variables shall not be described more than once in a sequence of actual arguments or binary operation expressions.	12.2	R13.2	5-0-1
	R3.6.3	sizeof operator shall not be used in expressions that have side effect.			
R3.7 Pay attention to the behavior of classes.	R3.7.1	In the class that manages resources, copy constructor, copy assignment operator and destructor shall be defined.			
	R3.7.2	Virtual destructor shall be declared in the base class.			
	R3.7.3	Copy assignment operator and move assignment operator shall be defined to comply with the following rules that state that: 1. Copy assignment operator and move assignment operator shall return a self-reference. 2. Copy assignment operator shall declare in the form of either "T&operator=(constT&)" or "T&operator =(T)". Move assignment operator shall declare in the form of "T &operator=(T &&)". Their return type shall not be const-qualified. 3. Copy assignment operator shall be capable of self-assignment.			
	R3.7.4	Default parameter values shall not be changed when overriding virtual functions.			8-3-1
	R3.7.5	Non-virtual function shall not be redefined in the derived class.			
	R3.7.6	Pass by reference or pass by pointer shall be used to set an object for polymorphic behavior as the function argument.			
	R3.7.7	Override keyword shall be written for overriding of virtual function to occur.			

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
R3.8 Pay attention to the behavior of exceptions.	R3.8.1	(1) Exception handling shall not be used. (2) When using exception handling, 《its method of use in the project shall be specifically defined》 .			15-0-1
	R3.8.2	Exception specification shall not be described.			15-4-1 15-5-2
	R3.8.3	Null shall not be thrown.			15-1-2
	R3.8.4	Pointer shall not be thrown as an exception.			15-0-2
	R3.8.5	Destructor shall not throw exceptions.			15-5-1
	R3.8.6	No argument shall be written in the throw expression when rethrowing an exception.			
	R3.8.7	Exception object shall be caught by reference.			15-3-5
	R3.8.8	Exception handlers shall be written in the order of derived class, base class and "… " (that catches all the exceptions).			15-3-6 15-3-7
	R3.8.9	All the exceptions shall be caught without any omission in the main function and thread start function.			15-3-2
R3.9 Pay attention to the behavior of templates.	R3.9.1	In case the template formal parameter is referenced by pointer, template specialization shall be prepared.			
R3.10 Pay attention to the behavior of lambda expressions.	R3.10.1	In lambda expressions, default capture mode shall not be used, and all the local names used shall be written explicitly.			
R3.11 Be careful with how to access the shared data in programs that use threads or signals.	R3.11.1	std::atomic shall be used for parallel concurrent processing instead of volatile.			
	R3.11.2	Bit fields that may be allocated in the same memory space shall not be accessed by multiple threads or shall be exclusively controlled properly.			
[Maintainability M1] Keep in mind that others will read the program.					
M1.1 Do not leave unused descriptions.	M1.1.1	Unused functions, variables, parameters typedefs, tags, labels or macros shall not be declared (defined).			0-1-3 0-1-5 0-1-10 0-1-11 0-1-12
	M1.1.2	(1) Sections of code should not be "commented out". (2) For commenting out sections of code, 《the coding rule shall be specified.》	2.4	D4.4	2-7-2 2-7-3
M1.2 Do not write confusingly.	M1.2.1	(1) Only one variable shall be declared in one declaration statement (avoid multiple declarations). (2) Automatic variables of the same type used for the similar purposes may be declared in one declaration statement, but variables with initialization and variables without initialization shall not be mixed.			
	M1.2.2	Suffixes shall be added to constant descriptions that can use them to indicate appropriate types. Only an uppercase letter "L" shall be used for a suffix indicating a long type integer constant.			2-13-3 2-13-4
	M1.2.3	When expressing a long string literal, successive string literals shall be concatenated without using newlines within the string literal.			
	M1.2.4	《A rule specifying how to use the namespace shall be defined.》			7-3-4
	M1.2.5	Namespace definition shall not be nested.			
	M1.2.6	Function template shall not be explicitly specialized.			14-8-1
	M1.2.7	If the declaration of a constructor becomes the same as the declaration of a default copy constructor when all the parameters specified with a default value are excluded, such constructor shall not be defined.			
M1.3 Do not write in an unconventional style.	M1.3.1	Expressions evaluating to true or false shall not be described in switch (expression).	15.4	R16.7	6-4-7
	M1.3.2	The case labels and default label in a switch statement shall be described only in the compound statement (excluding nested compound statements) within the body of the switch statement.	15.1	R16.2	6-4-4
	M1.3.3 (C edition)	The types shall be explicitly described for definitions and declarations of functions and variables.			
M1.4 Write in a style that clearly specifies the order of evaluation of operations.	M1.4.1	Expressions described at the right hand and left hand of && and || operations shall be either expressions that do not include binary operation or expressions enclosed with ( ). However, if only && operations or only || operations are successively combined, it is not necessary to enclose each && and || expression with ( ).	12.5	R12.1	5-0-2 5-2-1
	M1.4.2	《Usage of parentheses to explicitly indicate operator precedence shall be defined.》	12.1	R12.1	5-0-2
M1.5 Explicitly describe the operations that are likely to cause misunderstanding when they are omitted.	M1.5.1	A function identifier (function name) shall only be used with either a preceding "& " or with a parenthesized parameter list, which may be empty.	16.9		8-4-4
	M1.5.2	The conditional expression in an if statement or loop shall explicitly state that the type is bool .	13.2	R14.4	5-0-13
M1.6 Use one area for one purpose.	M1.6.1	Variables shall be prepared for each purpose.	18.3		
	M1.6.2	(1) Unions shall not be used. (2) If unions are used, the same members that are assigned values shall be referenced.	18.4	R19.2	9-5-1

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
M1.7 Do not reuse names.	M1.7.1	The following rules shall be followed to ensure name uniqueness: 1. An identifier declared in an inner scope shall not hide an indentifier declared in an outer scope. 2. A typedef name (including qualification, if any) shall be a unique identifier. 3. A tag name, union name and or enumeration name (including qualified names, if any) shall all be a unique identifier. 4. No object or function identifier with static storage duration should be reused. 5. No identifier in one category should have the same spelling as an identifier in another category.	5.2 5.3 5.4 5.5 5.6	R5.3 R5.6 R5.7 R5.8 R5.9	2-10-2 2-10-3 2-10-4 2-10-5 2-10-6
	M1.7.2	Names of functions, variables and macros in the standard library shall not be redefined or reused. In addition, these macro names shall not be undefined.	20.1 20.2	R21.1 R21.2	17-0-1 17-0-2 17-0-3
	M1.7.3	Names (variables) that start with an underscore shall not be defined.			17-0-1 17-0-2
M1.8 Do not use language specifications that are likely to cause misunderstanding.	M1.8.1	The right-hand operand of a logical && or || operator shall not contain side effects.	12.4	R13.5	5-14-1
	M1.8.2	(1) Macros shall only expand to a braced initializer, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-whilezero construct.. (2) Macros shall only expand to a guard for prevention of redundant inclusion of the header file, a type qualifier, or a storage class specifier.	19.4	R20.4	16-2-2
	M1.8.3	#line shall not be used, unless it is automatically generated by a tool.			
	M1.8.4	Sequences of three or more characters starting with ?? and alternative tokens shall not be used.	4.2	R4.2	2-3-1 2-5-1
	M1.8.5	A sequence starting with zero (0) that is two or more digits long shall not be used as a constant.	7.1	R4.1 R7.1	2-13-2
	M1.8.6	&& || , (comma) and & operators shall not be overloaded.			5-2-11 5-3-3
	M1.8.7	explicit specifier shall be used for conversion function.			
	M1.8.8	Single-parameter constructor shall have an explicit specifier.			12-1-3
M1.9 When writing in an unconventional style, explicitly state its intention.	M1.9.1	If statements that do nothing need to be intentionally described, comments or empty macros shall be used to make them noticeable.	14.3	R15.6	6-2-3
	M1.9.2	《The unified style of writing infinite loops shall be defined.》			
M1.10 Do not embed magic numbers.	M1.10.1	A meaningful constant shall be used after defining it as a constant with a name.			
M1.11 Explicitly state the area attributes.	M1.11.1	Read-only areas shall be declared as const type.	16.7	R8.13	7-1-1 7-1-2
	M1.11.2	Areas that may be updated by other execution units shall be declared as volatile .			
	M1.11.3	《Rules for variable declaration and definition for ROMization shall be defined》 .			
M1.12 Correctly describe the statements even if they are not compiled.	M1.12.1	Correct code shall be described even if it is going to be deleted by the preprocessor.	19.16	R20.13	
[Maintainability M2] Write in a style that can prevent modification errors.					
M2.1 Clarify the grouping of structured data and blocks.	M2.1.1	If arrays and structures are initialized with values other than 0, their structural form shall be indicated by using braces '{ } '. Data shall be described without any omission, except when all values are 0.	9.2	R9.2 R9.3	8-5-2
	M2.1.2	The body of if , else if , else , while , do , for , and switch statements shall be enclosed into blocks.	14.8 14.9	R15.6	6-3-1 6-4-1
M2.2 Localize access ranges and related data.	M2.2.1	Variables used only in one function shall be declared within the function.	8.7	R8.5	
	M2.2.2	Variables accessed by several functions defined in the same file shall be declared by either one of the following methods so that they will become inaccessible from other files: (1) By declaring these variables outside of function, using the static storage class specifiers; (2) By declaring these variables in unnamed namespace.	8.10	R8.7 R8.8	
	M2.2.3	Functions called from only the functions defined in the same file shall be declared by either one of the following methods so that they will not be called from other files: (1) By declaring these functions as static functions; (2) By declaring these functions in unnamed namespace.	8.10 8.11		
	M2.2.4	enum shall be used when defining related constants.			
	M2.2.5	Data members shall be private.			11-0-1
M2.3 Commonalize the code used to do the same process.	M2.3.1	A constructor shall be used for object initialization processed in the same way.			

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
[Maintainability M3] Write programs simply.					
M3.1 Do structured programming.	M3.1.1	For any iteration statement, there shall be at most one break statement or goto statement used for loop termination.	14.6	R15.4	6-6-4
	M3.1.2	(1) The goto statement shall not be used. (2) When using a goto statement, the destination to jump to shall be the label declared after the goto statement that is within the block enclosing the goto statement.	14.4	R15.1 R15.2 R15.3	6-6-2
	M3.1.3 (Vacant)	The continue statement shall not be used.	14.5		6-6-3
	M3.1.4	(1) Each case clause and default clause in a switch statement shall always end with a break statement. (2) If the case clause or default clause in a switch statement is not going to be ended with a break statement, 《a project-specific comment shall be defined》 and that comment shall instead be inserted.	15.2	R16.3	6-4-5
	M3.1.5	(1) A function shall end with one return statement. (2) A return statement to return in the middle of processing shall be written only in case of recovery from abnormality.	14.7	R15.5	6-6-5
M3.2 Limit the number of side effects per statement to one.	M3.2.1	(1) Comma expressions shall not be used. (2) Comma expressions shall not be used, other than in expressions for initializing or updating in for statements.	12.10	R12.3	5-18-1
	M3.2.2	Multiple assignments shall not be written in one statement, except when the same value is assigned to multiple variables.			
M3.3 Write expressions that differ in purpose separately.	M3.3.1	The three expressions of a for statement shall be concerned only with loop control.	13.5	R14.2	6-5-3 6-5-5
	M3.3.2	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	13.6	R14.2	
	M3.3.3	(1) Assignment operators shall not be used in expressions to examine true or false. (2) Assignment operators shall not be used in expressions to examine true or false, except for conventionally used notations.	13.1	R13.4	6-2-1
M3.4 Do not use complicated pointer operation.	M3.4.1	Three or more pointer indirections shall not be used.	17.5	R18.5	5-0-19
M3.5 Do not use complicated class structure.	M3.5.1	Virtual inheritance and non-virtual inheritance shall not be mixed in an accessible base class in the same hierarchical structure.			10-1-3
[Maintainability M4] Write in a unified style.					
M4.1 Unify the coding styles.	M4.1.1	《Conventions regarding the style of using, such as, the braces'{ } ', indentation and space shall be defined.》			
	M4.1.2	C++ style casting shall be used, provided that casting to void shall be allowed.			5-2-4
M4.2 Unify the style of writing comments.	M4.2.1	《Convention regarding the style of writing file header comments, function header comments, end of line comments, block comments and copyright shall be defined.》			
M4.3 Unify the naming convention.	M4.3.1	《Convention for naming external variables and internal variables shall be defined.》			
	M4.3.2	《Convention for naming files shall be defined.》			
M4.4 Unify the contents to be described in a file and the order of describing them.	M4.4.1	《The descriptive contents of header files (declarations, definitions, etc) and the order they are described in shall be defined.》			
	M4.4.2	《The descriptive contents of source files (declarations, definitions, etc) and the order they are described in shall be defined.》	8.6		
	M4.4.3	To use or define external variables or functions (except for functions used only in the file), the header file describing their declarations shall be included.	8.8	R8.5	3-3-1
	M4.4.4 (C edition)	External variables shall not be defined in multiple locations.			
	M4.4.5 (C edition)	Variable definitions or function definitions shall not be described in a header file.			
	M4.4.6	Header files shall be descriptively capable of handling redundant inclusions. 《The descriptive method to achieve this capability shall be defined.》	19.15	D4.10	16-2-3
	M4.4.7	using directive and using declaration of the namespace shall not be written before #include in the source file or in the header file, except in case of writing using declaration in class scope or function scope.			7-3-5 7-3-6
	M4.4.8	《The order of writing the class members shall be defined.》			

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
M4.5 Unify the style of writing declarations.	M4.5.1	(1) In a function prototype declaration, all the parameters shall not be named (types only) . (2) In a function prototype declaration, all the parameters shall be named. In addition, the types of the parameters, their names and the type of the return value shall be literally the same as those of the function definition.	16.3 16.4	R8.2 R8.3	3-9-1 8-4-2
	M4.5.2	Definition of class or enumeration and function declaration shall be performed separately.			
	M4.5.3	(1) "," shall not be placed before the last "} " in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators. (2) "," shall not be placed before the last "} " in the list of initial value expressions for structures, unions and arrays, nor in the list of enumerators. However, placing "," before the last "} " in the list of initial values for array initialization is allowed.			
M4.6 Unify the style of writing null pointers.	M4.6.1	(1) nullptr shall be used for the null pointer. (2) 0 shall be used for the null pointer. NULL shall not be used in any case. (3) NULL shall be used for the null pointer. NULL shall not be used for anything other than the null pointer.			4-10-1 4-10-2
M4.7 Unify the style of writing preprocessor directives.	M4.7.1	The body and parameters of a macro that includes operators shall be enclosed with parentheses ( ) .	19.10	R20.7	16-0-6
	M4.7.2	#else, #elif or #endif that corresponds to #ifdef, #ifndef or #if shall be described in the same file, and《their correspondence relationship shall be clearly stated with a comment defined in the project》 .	19.17	R20.14	16-1-2
	M4.7.3	To check whether a macro name has already been defined or not with #if or #elif, defined (macro name) or defined macro name shall be used.			16-0-7
	M4.7.4 (Vacant)	defined operator used in #if or #elif shall be written as "defined (macro name)" or "defined macro name", and not in any other way.	19.14		16-1-1
	M4.7.5	Macros shall not be #define 'd or #undef 'd within a block.	19.5		16-0-2
	M4.7.6	#undef shall not be used.	19.6	R20.5	16-0-3
	M4.7.7	The controlling expression of a #if or #elif preprocessing directive shall be evaluated to 0 or 1.		R20.8	
M4.8 Unify the style of writing overloads.	M4.8.1	Operator overload shall be defined according to the original meaning of the operator and together with other operators in the same category.			5-17-1
	M4.8.2	When defining operators new and delete of a class, all the standard forms shall be defined.			
	M4.8.3	Operators new and delete shall be defined in pairs.			
[Maintainability M5] Write in a style that makes testing easy.					
M5.1 Write in a style that makes it easy to investigate the causes of problems when they occur.	M5.1.1	《The rules for writing the code for setting debug options and for recording logs in release modules shall be defined.》			
	M5.1.2	(1) The # and ## preprocessor operators should not be used. (2) A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.	19.13 19.12	R20.10 R20.11	16-3-1 16-3-2
	M5.1.3	Inline function shall be used rather than using function-like macro.	19.7	D4.9	16-0-4
M5.2 Be careful when using dynamic memory allocations.	M5.2.1	(1) Dynamic memory shall not be used. (2) If dynamic memory is used,《the maximum amount of memory that can be used, process to be taken when running out of memory, and debugging procedure shall be defined》 .	20.4	R21.3 D4.12	18-4-1
[Portability P1] Write in a style that is not dependent on the compiler.					
P1.1 Do not use functionalities that are advanced features or implementation-defined.	P1.1.1	(1) Functionalities not specified in the language standard shall not be used. (2) If functionalities not specified in the language standard are used,《the functionalities used and their usage shall be documented.》	1.1	R1.1 R1.2	1-0-1
	P1.1.2	《All usage of implementation-defined behavior shall be documented.》	3.1	D1.1	
	P1.1.3	To use a program written in another language,《its interface shall be documented and its usage shall be defined》 .	1.3	D1.1	1-0-2
P1.2 Use only the characters and escape sequences defined in the language standard.	P1.2.1	To use characters other than those defined in the language standard for writing a program, the compiler specifications shall be confirmed and《their usage shall be defined》 .	3.2	D1.1	2-2-1
	P1.2.2	Only escape sequences defined in the language standard shall be used.	4.1	R4.1	2-13-1

Practice in detail	Rule		Relationship with MISRA rules		
			C:2004	C:2012	C++:2008
P1.3 Confirm and document data type representations, behavioral specifications of advanced functionalities and implementationdependent parts.	P1.3.1	Simple char type (that does not specify the signedness) shall be used only for storing character values. If a process that depends on signedness (implementation-defined) is required, unsigned char or signed char that specifies its signedness shall be used.	6.1 6.2	R10.1 R10.2 R10.3 R10.4	5-0-11 5-0-12
	P1.3.2	The members of an enumeration (enum) type shall be defined with values that can be represented as int type.			
	P1.3.3	(1) Bit fields shall not be used. (2) Bit fields shall not be used for data whose bit positions are meaningful. (3) 《 If it is being relied upon, the implementationdefined behavior and packing of bit fields shall be documented.》	3.5	D1.1	9-6-1
P1.4 For source file inclusion, confirm the implementationdependent parts and write in a style that is not implementationdependent.	P1.4.1	The #include directive shall be followed by either a <filename> or “filename” sequence.	19.3	R20.3	16-2-6
	P1.4.2	《The usage of < > format and “ ” format for #include file specification shall be defined.》	19.3	R20.3	
	P1.4.3	Characters ‘, \, “, /*, // and : shall not be used for file specification in #include.	19.2	R20.2	16-2-4 16-2-5
P1.5 Write in a style that does not depend on the environment used for compiling.	P1.5.1	The absolute path shall not be written for #include file specification.			
P1.6 Do not use unrecommended functionalities.	P1.6.1	Deprecated functionalities shall not be used.			
[Portability P2] Localize the code that has a problem with portability.					
P2.1 Localize the code that has a problem with portability.	P2.1.1	When assembly language programs are called from C++ language, 《how to localize such parts shall be defined》 , such as, by expressing them as functions of C++ language that contain only inline assembly language code or describing them using macros.	2.1	D4.2 D4.3	7-4-3
	P2.1.2	Keywords extended by the processing system shall be used by localizing them after 《defining the macros》 .			
	P2.1.3	(1) The basic types (char , int , long , long long, float , and double) shall not be used. Instead, types created with typedef shall be used. 《The types defined by typedef that are used in the project shall be defined.》 (2) When using any of the basic types (char, int, long, long long, float, and double) in a form that is dependent on its size, the type defined by typedef for each of these basic types shall be used. 《The types defined by typedef that are used in the project shall be defined.》	6.3	D4.6	3-9-2
[Efficiency E1] Write in a style that takes account of resource and time efficiencies.					
E1.1 Write in a style that takes account of resource and time efficiencies.	E1.1.1	Macro functions shall be used only in parts related to speed performance.			
	E1.1.2	Operations that remain unchanged shall not be performed within an iterated process.			
	E1.1.3	If the function argument is type class, pass by reference or pass by pointer shall be used instead of pass by value.			
	E1.1.4	《The policy of selecting either switch or if statement shall be determined and defined by taking readability and efficiency into consideration.》			
	E1.1.5	Variable definition and function definition shall not be written in the header file.	8.5		3-1-1
	E1.1.6	For initialization of data members, initialization shall be written in member declaration or constructor initializer shall be used.			
	E1.1.7	Overload function that corresponds with the type shall be defined to prevent implicit type conversion from occurring.			
	E1.1.8	Prefix form shall be used for increment and decrement operators.			
	E1.1.9	The use of virtual inheritance shall be allowed only when the classes that have the same base class are multiply inherited.			10-1-2
	E1.1.10	Any code that is unrelated to the template parameter shall not be written in the template definition.			

Citations and References

[1] ISO/IEC 25010:2011, Systems and software engineering-Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models

[2] THE CERT C SECURE CODING STANDARD 1st Edition, Robert C. Seacord, ISBN-13:978-0321563217, Addison-Wesley Professional, October 2008

[3] ISO/IEC 9899:1999, Programming languages – C, ISO/IEC 9899/Cor1:2001

[4] ISO/IEC 14882:2003, Programming languages -- C++

[5] ISO/IEC 14882:2011, Programming languages -- C++

[6] ISO/IEC 14882:2014, Programming languages -- C++

[7] The C++ Programming Language Fourth Edition, Bjarne Stroustrup, Addison-Wesley Professional, ISBN-13: 978-0321563842, May 2013

[8] The C Programming Language, Second Edition", Brian W. Kernighan, Dennis M. Ritchie, ISBN-13: 978-0131103627, Prentice Hall, March 1988

[9] Guidelines for the Use of the C Language in Vehicle Based Software, The Motor Industry Software Reliability Association, ISBN-13: 978-0952415664, April 1998, www.misra-c.com

[10] Guidelines for the Use of the C language in Critical Systems, The Motor Industry Software Reliability Association, ISBN-13: 978-0952415626 (paperback), October 2004, www.misra-c.com

[11] Guidelines for the Use of the C language in Critical Systems, The Motor Industry Software Reliability Association, ISBN-13: 978-1906400101 (paperback), March 2013, www.misra-c.com

[12] Guidelines for the Use of the C++ Language in Critical Systems, The Motor Industry Software Reliability Association, ISBN-13: 978-906400033 (paperback), June 2008, www.misra-cpp.com

[13] MISRA-C : Guidelines for programming of highly reliable software for embedded system developers, SESSAME Working Group 3, ISBN-13: 978-4542503342, Japanese Standards Association, May 2004 (in Japanese)

[14] Indian Hill C Style and Coding Standards, ftp://ftp.cs.utoronto.ca/doc/programming/ihstyle.ps

[15] Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs, Steve Maguire, ISBN-13: 978-1556155512, Microsoft Press, May 1993

[16] The Practice of Programming, Brian W. Kernighan, Rob Pike, ISBN-13: 978-0201615869, Addison-Wesley Professional, February 1999

[17] C Style: Standards and Guidelines: Defining Programming Standards for Professional C Programmers, David Straker, ISBN-13: 978-0131168985, Prentice Hall, January 1992

[18] C Programming FAQs: Frequently Asked Questions, Steve Summit, ISBN-13: 9780201845198, Addison-Wesley Professional, November 1995

[19] JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM, Document Number 2RDU00001 Rev C, Lockheed Martin Corporation, December 2005, http://www.stroustrup.com/JSF-AV-rules.pdf

[20] Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition", Scott Meyers, ISBN-13: 978-0321334879, Addison-Wesley Professional, May 2005

[21] More Effective C++: 35 New Ways to Improve Your Programs and Designs, Scott Meyers, ISBN-13: 978-0201633719, Addison-Wesley Professional, January 1996

[22] Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, Scott Meyers, ISBN-13: 978-1491903995, Oreilly & Associates Inc, December 2014

[23] C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Herb Sutter, Andrei Alexandrescu, ISBN-13: 978-0321113580, Addison-Wesley Professional, November 2004

[24] GNU Coding Standards, Free Software Foundation, http://www.gnu.org/prep/standards/

[25] Linux kernel coding style, https://www.kernel.org/doc/Documentation/CodingStyle

[26] Google C++ Style Guide, Google, https://google.github.io/styleguide/cppguide.html

Ver.1.0 Authors and editors (in alphabetical order)

FUTAGAMI Takao	TOYO Corporation
HACHIYA Shouichi	GAIA System Solutions Inc.
HIRAYAMA Masayuki	IPA/SEC (Toshiba Corporation)
ITOH Masako	Fujitsu Limited
KOGA Keiko	Hitachi Solutions, Ltd.
MITSUHASHI Fusako	NEC Corporation
NISHIYAMA Hiroyasu	Hitachi, Ltd.
SASAKI Kouji	Fujitsu Software Technologies Limited
SHISHIDO Fumio	emBex Inc.
TOYAMA Keisuke	IPA/SEC (Hitachi, Ltd.)
UNO Musubi	Panasonic Corporation
YOSHIZAWA Satomi	NEC Corporation

Ver.2.0 Authors and editors (in alphabetical order)

FUJIMOTO Takanari	Mitsubishi Electric Corporation
FUTAGAMI Takao	TOYO Corporation
ITOH Masako	Fujitsu Limited
MIHARA Yukihiro	IPA/SEC (Debug Engineering Institute)
MITSUHASHI Fusako	NEC Corporation
NISHIYAMA Hiroyasu	Hitachi, Ltd.
OBATA Hiromi	IPA/SEC
SHUKUGUCHI Masahiko	eSOL Co., Ltd.
TACHI Nobuyuki	Nagoya University
TOYAMA Keisuke	IPA/SEC

(Organizational affiliations are as of the publication of Japanese edition.)

Editorial supervisor

NAKATA Ikuo	Professor Emeritus at the University of Tsukuba

Contributors to English translation version

SHIMIZU Tatsuo	Shimizu International, Inc.
TOYAMA Keisuke	IPA/SEC

Written and edited by Software Reliability Enhancement Center,
Technology Headquarters, Information-technology Promotion Agency, Japan

ESCR C++
【Revised Edition】
Embedded System development Coding Reference guide [C++ Language Edition]
Ver. 2.0

October 1, 2016

Written and edited by Software Reliability Enhancement Center,
Technology Headquarters, Information-technology Promotion Agency, Japan

Bunkyo Green Court Center Office
2-28-8 Honkomagome, Bunkyo-ku, Tokyo, 113-6591 Japan
http://www.ipa.go.jp/english/sec

IPA Information-technology Promotion Agency,Japan