

改訂版

組込みソフトウェア開発向け コーディング作法ガイド [C言語版]

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター 編

はじめに

ESCR Ver. 2.0 発行にあたり

本書はC言語を用いて開発されるソフトウェアのソースコードの品質をよりよいものとするを目的として、コーディングの際に注意すべきことやノウハウを「組込みソフトウェア向けコーディング作法ガイド」(英語名 = ESCR: Embedded System development Coding Reference) として整理したものです。

ESCRは2006年6月にVer. 1.0をリリースし、利用者の皆様からの指摘事項や一部誤り、説明などで分かりにくいところを修正し2007年6月にVer. 1.1として改訂版をリリースしています。Ver. 1.1までは、C言語規格として最も一般的に普及していたC90に準拠していましたが、昨今後継の規格であるC99が使われるようになりつつあります。この状況に対応するべくMISRA Cが2013年3月に大幅に改訂(MISRA C:2012)されました。そこで今回、以下の二点を目的にVer. 2.0としてESCRを新たに改訂しました。

- ・準拠する言語規格をJIS最新のC99とし、新機能を利用し易いルールとする
- ・相互関係にあるMISRA Cの大幅な改訂に合わせ、旧版の参照箇所を見直して整合性をとる

Ver. 1.1からの継続性を保つために、作法やルール番号は変更せず、C99で拡張された言語仕様に関するものについてルールの追加あるいはルールの解説文や適合例 / 不適合例などの追加を行っています。さらにMISRA Cの参照に関して、MISRA C:2012で変更・追加のあったルールについてはMISRA C:2012のルール番号を、旧版にしか存在しなくなったルールについては旧版(MISRA C:2004)のルール番号を記載しています。

ESCR Ver. 2.0は、従来通り「C言語などを用いて組込みソフトウェアを作成する場合に、ソースコードの標準化や品質の均一化を進めることを目的として組織やグループ内のコーディングルールを決める際の参考として利用していただく」ことを目的としています。また旧版と同じく、以下に示す三つのパートの構成としています。

Part 1. コーディング作法ガイドの読み方

Part 2. 組込みソフトウェア向けコーディング作法：作法表

Part 3. 組込みソフトウェアにありがちなコーディングミス

本書は、C言語規格C99、MISRA C:2004 / MISRA C:2012をベースに、コーディング作法改訂WGメンバーの協力によりESCR Ver. 1.1を精査し、見直したものです。引き続き本書を有効に活用いただき、組込みソフトウェアの生産性向上、及び高品質なソフトウェア開発を実現していただくことを願っています。

2014年春

独立行政法人 情報処理推進機構 技術本部 ソフトウェア高信頼化センター

三原 幸博、十山 圭介

コーディング作法ガイド改訂WG

三橋 二彩子

目次

はじめに

Part 1 コーディング作法ガイドの読み方 1

1 概要	2
1.1 コーディング作法とは	2
1.2 コーディング作法の目的と位置付け・想定利用者	3
1.3 コーディング作法の特徴	4
1.4 本ガイド利用に関する注意事項	5
2 ソースコード品質のとらえ方	8
2.1 品質特性	8
2.2 品質特性と作法・ルールの考え方	11
3 本ガイドの利用方法	13
3.1 本ガイドの利用シーン	13
3.2 新規コーディング規約の作成	14
3.3 既存コーディング規約の充実	16
3.4 プログラマの研修、独習のための学習教材	17

Part 2 組込みソフトウェア向けコーディング作法：作法表 19

作法表の読み方	20
作法表中の用語	23
組込みソフトウェア向けコーディング作法	24
●信頼性	25
●保守性	63
●移植性	121
●効率性	135

Part 3 組込みソフトウェアにありがちなコーディングミス 139

組込みソフトウェアにありがちなコーディングミス	140
1 意味のない式や文	140
2 誤った式や文	142
3 誤ったメモリの使用	143
4 論理演算の勘違いによる誤り	146
5 タイプミスによる誤り	147
6 コンパイラによってはエラーにならないケースがある記述	147

付録 149

付録 A 作法・ルール一覧	151
付録 B C 言語文法によるルール分類	163
付録 C 処理系定義の動作について	173

引用・参考文献	180
---------------	-----

Part 1

コーディング 作法ガイドの読み方

1 概要

- 1.1 コーディング作法とは
- 1.2 コーディング作法の目的と位置付け・想定利用者
- 1.3 コーディング作法の特徴
- 1.4 本ガイド利用に関する注意事項

2 ソースコード品質のとらえ方

- 2.1 品質特性
- 2.2 品質特性と作法・ルールの考え方

3 本ガイドの利用方法

- 3.1 本ガイドの利用シーン
- 3.2 新規コーディング規約の作成
- 3.3 既存コーディング規約の充実
- 3.4 プログラマの研修、独習のための学習教材

1 概要

1.1 コーディング作法とは

組込みソフトウェアを作る上でソースコードを作成する作業（コード実装）は避けて通ることができません。この作業の出来不出来はその後のソフトウェアの品質を大きく左右します。一方で、組込みソフトウェア開発で最も多く利用されているC言語の場合、記述の自由度が高く、技術者の経験の差が出やすい言語と言われています。技術者の技量や経験の差によって、作られるソースコードの出来不出来に差が生じてしまうのは好ましくありません。先進的な企業の中にはこうした事態を防ぐために、組織として、あるいはグループとして守るべきコーディング基準やコーディング規約を定め、ソースコードの標準化を進めているケースもあります。



コーディング規約に関する課題点

通常、コーディング規約とは「品質を保つために守るべきコードの書き方（ルール）」を整理したものとなっていますが、現在利用されているコーディング規約に関しては、下記のような課題が存在しています。

- 1) ルールの必要性が理解されない。または、ルール違反に対する正しい対処方法が理解されていない。
- 2) ルールが多すぎて覚えきれない。あるいは、ルールが少なくてカバー範囲が不足している。
- 3) ルールの遵守状況を確認するための高精度のツールがなく、確認を技術者が目視で行うレビューに頼っており負担が大きい。

また、この結果として、すでにコーディング規約がある組織や部門においても、それらが形骸化して守られていないといった状況も散見されます。

さらに、どのような形であれコーディング規約が用意されていればまだよく、コーディング規約自体が決められずに、依然として個々の担当者の判断に任せたコーディングが中心となっている組織も少なくありません。



コーディング作法とは

本ガイドで提供する「コーディング作法」とは、このようなコーディング規約に関する現場の問題を解決することを目的として、様々なコーディングのシーンで守るべき基本的な考え方（基本概念）をソフトウェアの品質の視点を考慮して「作法」として整理したものです。本ガイドではこうした「作法」とこれに関連するコーディング規約（ルール）の参考例を提示しています。

本書の利用者は、これらの情報を参考に、「自部門における具体的なコーディング規約を策定する」といった作業を行うことで、前述したコーディング規約に関する課題を解決することができます。

1.2 コーディング作法の目的と位置付け・想定利用者



コーディング作法の目的と位置付け

本ガイドは、企業やプロジェクトでコーディング規約を作成・運用する人に対して、コーディング規約作成の支援を目的としたコーディング作法ガイドです。本ガイドの特徴は、コーディング規約を「品質を保つために守るべきコードの書き方」と考え、ルールの基本概念を作法としてまとめたことです。作法は『JIS X 0129-1 ソフトウェア製品の品質 第1部：品質モデル』に準拠した品質概念を基に、作法概要、作法詳細に分類・階層化しています。さらに、それぞれの作法にC言語に対応したルールをその必要性とともに提示しています。この作法とルールにより、意義・必要性を理解できる、実用的な「コーディング規約」が容易に作成できることを目標としています。



想定する利用者

本ガイドは下記の利用者を想定して作成されています。

コーディング規約を作成する人

本ガイドを参考にして新規のコーディング規約を作成することができます。または既にあるコーディング規約の確認、整理ができます。

プログラマやプログラムレビューをする人

本ガイドの作法・ルールを理解・修得することによって、信頼性の高い、保守しやすいコードの作成が無理なくできるようになります。



得られる効果

本ガイドを利用することで直接的には前述のような効果を期待できます。さらにこの結果として、

- ・ ソフトウェアの品質面で大きなネックとなっている実装面での技術者による出来不出来のばらつきを解消できる
- ・ ソースコード上の明らかな誤りなどをコーディング段階やその後のレビューなどで早期に除去することができる

といった効果が期待できます。

1.3 コーディング作法の特徴

本ガイドで提供するコーディング作法は下記のような特徴をもっています。

体系化された作法・ルール

本ガイドでは、ソフトウェアの品質と同様に、コードの品質も「信頼性」「保守性」「移植性」などの品質特性で分類できると考え、コーディングの作法とルールを『JIS X 0129-1 ソフトウェア製品の品質』を基に体系化しています。本ガイドにおける作法とは、ソースコードの品質を保つための慣習や実装の考え方で、個々のルールの基本的な概念を示します。ルールについては、世の中に存在する多くのコーディング規約を十分に吟味し、現在の状況（言語仕様や処理系の実情）に合わせて取捨選択し、作法に対応させる形で示しています。作法とルールを品質特性で分類することで、それらがどの品質を保つことを主たる目的としているのかを理解できるようにしています。

なお、本ガイドが参照したコーディング規約として、本ガイドを検討したメンバーが所属する会社のコーディング規約、『MISRA-C』『Indian Hill C Style and Coding Standards』『GNU

coding standards』などがあります。詳細は巻末「引用・参考文献」をご参照ください。

すぐ使えるリファレンスルール

本ガイドでは、コーディング規約作成のための参考情報として、具体的なC言語用のルールを示しています。このルールはそのまま規約に利用することができます。後述の「3 本ガイドの利用方法」を参考に、必要なルールを選択し、その上で足りないルールを追加することで、C言語のコーディング規約を容易に作成することが可能です。

ルールの必要性を提示

本ガイドでは、ルールの必要性を、対応する作法及びルールの例と備考の説明で示しています。また、熟練したプログラマには当たり前と思われるルールは選択指針にそのことを示しています。必要性を考える上で参考にしてください。

他のコーディング規約との対応関係を明示

本ガイドでは、各ルールについて、世の中で使われているコーディング規約との対応関係を示しています。それによって、包含関係などを確認しやすくしています。対応を示しているコーディング規約としては、『MISRA-C』『Indian Hill C Style and Coding Standards』などがあります。

1.4 本ガイド利用に関する注意事項

本ガイドの利用に際しては下記のような点に注意してください。



ルールの範囲

本ガイドでは、次に関するルールは、C言語のリファレンスルールの対象外としています。

- ・ ライブラリ関数
- ・ メトリクス（関数の行数・複雑度など）
- ・ コーディングミスに分類されると思われる記述誤り

なお、最後の「コーディングミスに分類されると思われる記述誤り」に関しては、リファレンスルールから外していますが、今回のガイドを作成するにあたり、集められたコーディングミス例を、「Part3 組込みソフトウェアにありがちなコーディングミス」にまとめています。C言語を習得したての方が陥りやすいミスであり、C言語の初心者の方には参考になりますので一読ください。また、プロジェクトによっては、このようなコーディングミスの記述誤りについてもルール化した方がよいと判断するケースもあります。この場合は、Part3の例をもとに、ルール化を検討ください。



本ガイドで引用・参照している規格類について

本ガイドでは、以下の規格を引用・参照しています。

C90

『JIS X 3010:1996 プログラム言語C』で規定されるC言語規格のこと。『JIS X 3010:1993 プログラム言語C』が1996年に追補・訂正されたものである。翻訳元となるISO/IEC 9899:1990が1990年に発行されたため、「C90」と呼ぶことが多い。

C99

『JIS X 3010:2003 プログラム言語C』で規定されるC言語規格のこと。現状、世の中に普及しているC言語の規格である。翻訳元となるISO/IEC 9899:1999が1999年に発行されたため、「C99」と呼ぶことが多い。

C11

C99の後継として、2011年に制定されたISO/IEC 9899:2011で規定されるC言語規格のこと。C言語の最新の規格である。「C11」と呼ぶことが多い。

C++

『JIS X 3014:2003 プログラム言語C++』で規定されるC++言語規格のこと。

MISRA C

英国 The Motor Industry Software Reliability Association (MISRA) によって定められた、C 言語のコーディングガイドライン MISRA C:1998、MISRA C:2004、及び MISRA C:2012 のこと。

MISRA C:1998

引用・参考文献[5]の規約のこと。

MISRA C:2004

引用・参考文献[6]の規約のこと。MISRA C:1998 の改訂版である。

MISRA C:2012

引用・参考文献[6]の規約のこと。MISRA C:2004 の改訂版である。



変数名・関数名の付け方について

本書中に例として表記したコード中の変数名、関数名などは、対象とするルールの理解の妨げにならないよう、極力簡潔な表記を用いています。



Ver. 1.1 からの変更点について

本書では Ver. 1.1 からの変更点として、C99 の新機能を利用する場合に必要であると考えられるルールを追加するとともに、MISRA C:2012 の改訂内容と整合性を保つように一部のルールや解説を修正しています。ルールとしなくてもよいと判断して削除したルールもあり、また旧版と同様の内容でもより分かりやすく修正したものもあります。

削除したルール番号については欠番とし、旧版とこの改訂版で同じ内容のルールは同じ番号になり、これまでの ESCR 利用者の方も自然に利用できるものと考えています。

追加になったルールは、R1.3.4、R3.1.3、R3.1.4、M4.7.7 で、削除して欠番となったルールは、M3.1.3、M4.7.4 です。

2 ソースコード品質のとらえ方

2.1 品質特性

ソフトウェアの品質というと、一般的に「バグ」を思い浮かべる方が少なくないかと思います。しかし、ソフトウェア・エンジニアリングの世界では、ソフトウェアの製品としての品質はより広い概念でとらえられています。このソフトウェア製品の品質概念を整理したものが、ISO/IEC 25010であり、これをJIS化したものがJIS X 25010です。



JIS X 25010とソースコードの品質

JIS X 25010では、ソフトウェア製品の品質に関わる特性（品質特性）に関しては、「信頼性」「保守性」「移植性」「効率性」「セキュリティ」「機能性」「使用性」「互換性」の8つの特性を規定しています。

このうち、「機能性」と「使用性」、「互換性」の3特性については、より上流の設計段階以前に作り込むべき特性と考えられます。これに対し、ソースコード段階では「信頼性」「保守性」「移植性」「効率性」の4特性が深く関係すると考えられます。「セキュリティ」は、ソフトウェア製品の品質に関する旧規格（JIS X 0129-1）では「機能性」に含まれていた副特性であり、基本的には設計段階の特性と考えられますが、バッファオーバーフローを避けるなどセキュリティに影響するコーディングもあります。セキュリティに関連するコーディング作法に関しては、「CERT C セキュアコーディングスタンダード」を参照してください。

このため、本ガイドで提供する作法については、その大分類として、これら「信頼性」「保守性」「移植性」「効率性」の4特性を採用しています。表1に、本ガイドと関連するJIS X 25010の「品質特性」と本ガイドが考える「コードの品質」の関係を「品質副特性」とともに示します。

表1 ソフトウェアの品質特性とコードの品質

品質特性 (JIS X 25010)		品質副特性 (JIS X 25010)		コードの品質
信頼性	明示された時間帯で、明示された条件下に、システム、製品又は構成要素が明示された機能を実行する度合い。	成熟性	通常の運用操作の下で、システム、製品又は構成要素が信頼性に対するニーズに合致している度合い。	使い込んだときのバグの少なさ。
		可用性	使用することを要求されたとき、システム、製品又は構成要素が運用操作可能及びアクセス可能な度合い。	
		障害許容性 (耐故障性)	ハードウェア又はソフトウェア障害にもかかわらず、システム、製品又は構成要素が意図したように運用操作できる度合い。	バグやインターフェース違反などに対する許容性。
		回復性	中断時又は故障時に、製品又はシステムが直接的に影響を受けたデータを回復し、システムを希望する状態に復元することができる度合い。	
保守性	意図した保守者によって、製品又はシステムが修正することができる有効性及び効率性の度合い。	モジュール性	一つの構成要素に対する変更が他の構成要素に与える影響が最小になるように、システム又はコンピュータプログラムが別々の構成要素から構成されている度合い。	コードの一つの構成要素に対する変更が他の構成要素に与える影響が最小になるように構成されている度合い。
		再利用性	一つ以上のシステムに、又は他の資産作りに、資産を使用することができる度合い。	コードを他のプログラムに使用することができる度合い。
		解析性	製品若しくはシステムの一つ以上の部分への意図した変更が製品若しくはシステムに与える影響を総合評価すること、欠陥若しくは故障の原因を診断すること、又は修正しなければならない部分を識別することが可能であることについての有効性及び効率性の度合い。	コードの理解しやすさ。
		修正性	欠陥の取込みも既存の製品品質の低下もなく、有効的に、かつ、効率的に製品又はシステムを修正することができる度合い。	コードの修正しやすさ、修正による影響の少なさ。
		試験性	システム、製品又は構成要素について試験基準を確立することができ、その基準が満たされているかどうかを決定するために試験を実行することができる有効性及び効率性の度合い。	修正したコードのテスト、デバッグのしやすさ。

品質特性 (JIS X 25010)		品質副特性 (JIS X 25010)		コードの品質
移植性	一つのハードウェア、ソフトウェア又は他の運用環境若しくは利用環境からその他の環境に、システム、製品又は構成要素を移すことができる有効性及び効率性の度合い。	適応性	異なる又は進化していくハードウェア、ソフトウェア又は他の運用環境若しくは利用環境に、製品又はシステムが適応できる有効性及び効率性の度合い。	異なる環境への適応のしやすさ。 ※標準規格への適合性も含む。
		設置性	明示された環境において、製品又はシステムをうまく設置及び／又は削除できる有効性及び効率性の度合い。	
		置換性	同じ環境において、製品が同じ目的の別の明示された製品と置き換えることができる度合い。	
性能効率性	明記された状態（条件）で使用する資源の量に関する性能の度合い。	時間効率性	製品又はシステムの機能を実行するとき、製品又はシステムの応答時間及び処理時間、並びにスループット速度が要求事項を満足する度合い。	処理時間に関する効率性。
		資源効率性	製品又はシステムの機能を実行するとき、製品又はシステムで使用する資源の量及び種類が要求事項を満足する度合い。	資源に関する効率性。
		容量満足性	製品又はシステムのパラメータの最大限度が要求事項を満足させる度合い。	
セキュリティ	人間又は他の製品若しくはシステムが、認められた権限の種類及び水準に応じたデータアクセスの度合いをもてるように、製品又はシステムが情報及びデータを保護する度合い。	機密性	製品又はシステムが、アクセスすることを認められたデータだけにアクセスすることができることを確実にする度合い。	アクセスが認められたデータにだけアクセスすることができることを確実にする度合い。
		インテグリティ	コンピュータプログラム又はデータに権限をもたないでアクセスすること又は修正することを、システム、製品又は構成要素が防止する度合い。	プログラム又はデータ領域に権限をもたないでアクセスすること又は修正することを防止する度合い。
		否認防止性	事象又は行為が後になって否認されることがないように、行為又は事象が引き起こされたことを証明することができる度合い。	
		責任追跡性	実体の行為がその実体に一意的に追跡可能である度合い。	
		真正性	ある主体又は資源の同一性が主張したとおりであることを証明できる度合い。	

2.2 品質特性と作法・ルールの考え方



全体構造

本ガイドでは、ソースコードを作成する際に守るべき基本事項を「作法」として整理してあります。また、個々の「作法」に関してより具体的にコーディングの際に注意すべき事項を「ルール」として参考情報として紹介しています。

本ガイドでは、「作法」「ルール」を2.1に示した4つの品質特性に関連づけて分類・整理してあります。本ガイドにおける作法、ルールの意味は次の通りです（図1参照）。

作法

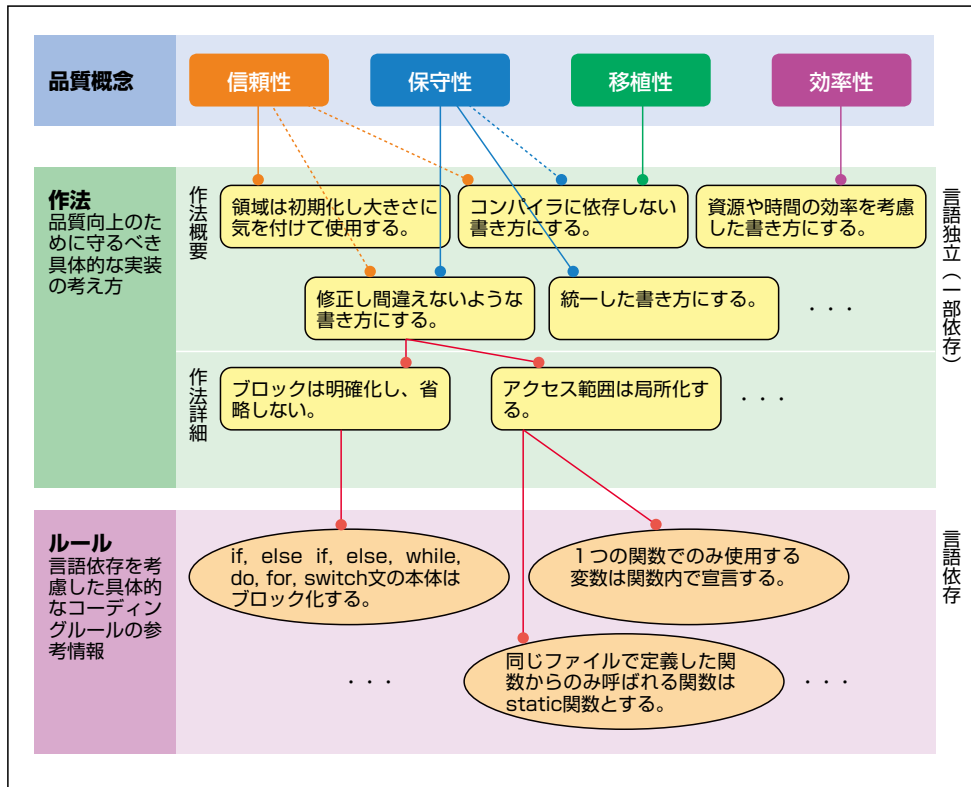
ソースコードの品質を保つための慣習、実装の考え方であり、個々のルールの基本概念を示します。作法概要、作法詳細に階層化して示してあります。

ルール

守らなければならない、具体的な一つひとつの決めごとであり、コーディング規約を構成します。本ガイドでは参考情報として示しています。なお、ルールの集まりもルールと呼ぶことがあります。

作法とルールの対応付け

作法、ルールの多くは、複数の品質特性と関連しますが、最も関連の強い特性に分類しています。品質特性と関連づけることにより、各作法がどのような品質に強く影響するかを理解できるようにしています。



ルールを参考にして作成

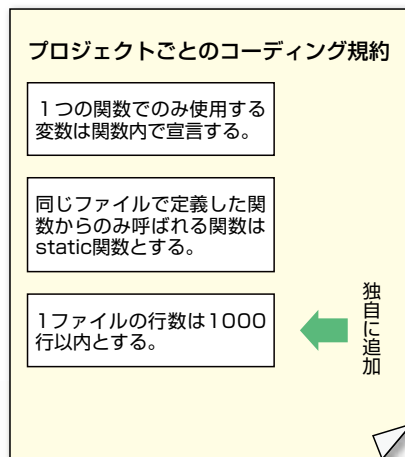


図1 品質概念、作法、ルールの関係

3 本ガイドの利用方法

3.1 本ガイドの利用シーン

想定する利用方法

本ガイドは、コーディング規約作成支援を目的とし、次の3通りの利用方法を想定しています。

- 1) 新規コーディング規約の作成
- 2) 既存コーディング規約の充実
- 3) プログラマの研修、独習のための学習教材

新規コーディング規約の作成

現在、組織や部門内で守るべきコーディング規約が整備できていない場合に、本ガイドを参考に、その部門に適したコーディング規約を作成することができます。

既存コーディング規約の充実

すでにコーディング規約が整備されている組織や部門であっても、それらを定期的にメンテナンスすることは有効です。その際に、本ガイドを参考にすることで、より効率的に既存のコーディング規約などの見直しをすることが可能になります。

プログラマの研修、独習のための学習教材

C言語に関する書籍は数多くあります。本ガイドはこうした既存の書籍とは異なり、実装面での品質という視点を前面に、品質を維持し向上するためのソースコードの作り方を整理してあります。その点で、ソースコードの品質について、より実践的な視点から学習をするための格好の教材として利用することも可能です。

3.2 新規コーディング規約の作成

ここではコーディング規約の存在しないプロジェクトが、本書を用いて新規にコーディング規約を作成する場合の手順を示します。



作成時期

コーディング規約は、プログラム設計に入る前までに作成します。コーディング規約は、コーディング時に参照するルール集ですが、関数名の命名規約などプログラム設計に関わるルールもあります。そのため、プログラム設計以前に作成する必要があります。



作成方法

新規にコーディング規約を策定する場合には、下記の順序で作成することをお勧めします。

Step-1 コーディング規約の作成方針を決定

Step-2 決定した作成方針に沿ってルールを選択

Step-3 ルールのプロジェクト依存部分を定義

Step-4 ルール適用除外の手順を決定

このあと、必要に応じてルールを追加してください。

Step-1 作成方針の決定

コーディング規約作成にあたっては、まず、コーディング規約の作成方針を決定します。コーディング規約の作成方針とは、プロジェクトが作成するソフトウェアやプロジェクトを構成する人の特性などから、そのプロジェクトで作成されるコードがどのような書き方となっているべきかを示す方針のことです。例えば、安全性を重視し、便利であっても危険な機能は使用しないという書き方にするのか、危険な機能であっても注意して使用する書き方にするのかなどが、この方針にあたります。なお、方針の決定にあたっては、プロジェクトで重視したい品質特性、及び次に示す視点を考慮してください。

- ・ フェールセーフを考慮したコーディング
- ・ プログラムを見やすくするコーディング
- ・ デバッグを考慮したコーディング

など

Step-2 ルールの選択

ルールは、Step-1 で決定した規約作成の方針に従い、Part2の作法表の中から選択します。例えば、移植性を重視する方針とした場合、移植性に該当するルールを多く選ぶなどの工夫をしてください。

本ガイドの「Part2 組込みソフトウェア向けコーディング作法」では、規約として採用しないとそのルールが属する品質特性を著しく損なうと考えられるルールについて「選択指針」欄の○で示しています。一方、言語仕様を熟知している人にはあえて規約にする必要がないと思われるルールについて●で示しています。これを参考にルールを選択してください。ルール選択の最も簡便な方法は、○が付いているルールのみを選択することです。それにより、ごく一般的なルールが選択できます。

Step-3 プロジェクト依存部分の定義

本ガイドのルールには、次の3種類のルールがあります。

- 1) 規約としてそのまま使えるルール（規約化欄にマークなしのルール）
- 2) プロジェクトの特性に合わせて、どのルールとするか選択する必要のあるルール（規約化欄に「選」印のルール）
- 3) 文書化により、規定する必要のあるルール（規約化欄に「規」または「文」印のルール）

2) と 3) のルールは、そのままではルールとして利用できません。2) のルールを採用した場合、提示されている複数のルールから、いずれかのルールを選択してください。3) のルールを採用した場合、各作法のページに補足として記載されている「ルール定義の指針」を参考に、ルールを規定してください。

Step-4 ルール適用除外の手順を決定

実現する機能により、コーディング時に注目すべき品質特性が異なる場合があります（例えば、保守性よりも効率性を重視しなければならないなど）。この場合、定めたルール通りに記述すると、目的が達せられないなどの不具合になる可能性があります。このような場合に対応するべく、部分的に、ルールを適用除外として認めることを手順化しておく必要があります。

重要なのは、ルール通りに記述することにより、どのような不具合になるかを明記し、これを有識者にレビューしてもらい、その結果を記録に残すことです。安易にルール適用除外を許してしまい、ルールが形骸化するのを防止してください。

以下に、適用除外を認める手順の例を示します。

【手順例】

(1)適用除外の理由書を作成する。

（理由書の項目例：「ルール番号」「発生箇所（ファイル名、行番号）」「ルール遵守の問題点」「ルール逸脱の影響など」）

(2)有識者のレビューを受ける。⇒レビュー結果を理由書に追記する

(3)コーディング工程の責任者の承認を受ける。⇒承認記録を理由書に記載する。

3.3 既存コーディング規約の充実

本ガイドは、コーディング規約が既に存在するプロジェクトに対しては、既存のコーディング規約をさらに充実したものとするための参考書として利用できます。



抜けモレの防止

本ガイドの作法概念を用いて、既存のコーディング規約をカテゴライズすることにより、抜けている観点を充実させたり、自プロジェクトが何に重点を置いて作業していたか再認識することができます。



ルール必要性の明確化

本ガイドの作法とルールの適合例などを参照することで、理由も分からず強制されていた規則の必要性を認識するためのツールとして利用できます。

3.4 プログラマの研修、独習のための学習教材

本ガイドは、C言語を一応勉強したが、実際のコーディングには不慣れ、経験が浅いなどのプログラマにとって格好の学習教材となります。



対象者

本ガイドは以下のプログラマを対象としています。

- ・C言語を一通り学習したプログラマ
- ・他言語でのプログラミング経験はあるが、C言語を使うのは初めてというプログラマ



学習できること

信頼性・保守性・移植性などの観点から分類された本ガイドを読むことにより、

- ・信頼性を高くするコーディング方法
- ・バグを作り込まないようにするコーディング方法
- ・デバッグ・テストがしやすいようにするコーディング方法
- ・他人が見て、見やすいようにするコーディング方法とその必要性

などを学習できます。

Part 2

組込みソフトウェア向け コーディング作法：作法表

■ 作法表の読み方

■ 作法表中の用語

■ 組込みソフトウェア向けコーディング作法

- 信頼性
- 保守性
- 移植性
- 効率性

作法表の読み方



作法の整理構造

Part2に示すコーディング作法は、ソフトウェア品質特性の中の4特性(信頼性、保守性、移植性、効率性)に従ってカテゴライズされています。

作法概要

各特性に深く関係する作法をさらに作法概要として整理してあります。例えば、保守性については、「保守性1：他人が読むことを意識する」から「保守性5：試験しやすい書き方にする」までの5つの作法概要に分けて整理してあります。

作法詳細

各作法概要内には、その作法概要に属する複数の作法(作法詳細)を整理してあります。例えば、「保守性3：プログラムはシンプルに書く」という作法概要については、

- 保守性 3.1 構造化プログラミングを行う。
- 保守性 3.2 1つの文で1つの副作用とする。
- 保守性 3.3 目的の違う式は、分離して記述する。
- 保守性 3.4 複雑なポインタ演算は使用しない。

といった4つの作法詳細が整理されています。



作法表の構成

個々の作法について、実際のコーディングの際に注意すべきルールの参考を表に整理してあります。表の各欄は次のような情報を掲載してあります。

④ ルール

作法に対応する具体的に守らなければならないC言語用のリファレンスルールです。なお、この欄のルールがMISRA Cからの引用である場合は、次の形式で示しています。

例：【MISRA C:2004 1.3】、【MISRA C:2012 R8.14】

⑤ 選択指針

本ガイドを用いてコーディング規約を作成する際のルールの選択指針です。

マークなし プロジェクトの特性に合わせて選択すればよいと思われるルール。

- 言語仕様を熟知している人にはあえて規約にする必要がないと思われるルール。(経験のあるプログラマには当たり前のこと)
- 守らないと著しく品質特性を損なうと考えられるルール。

⑥ 規約化

対象ルールが、プロジェクトごとの指針によって詳細を定める必要があるかないかを示しています。また、ルールとしてはそのまま利用できるが、例えば「コンパイラ依存の言語仕様の動作と使い方を文書として残す」などのように、文書作成を指示するルール(文書化ルールと呼ぶ)もこの欄で示します。

マークなし 詳細を定めたり、文書化したりする必要がないもの。

- 選** 選択。複数のルールが提示されており、その中から選択する必要があります。選択肢は、括弧付き数字((1)、(2)…)などで示しています。
- 規** プロジェクトごとに具体的なルールを規定する必要がある。規定すべき部分は《 》で囲んで明示しています。
- 文** 文書作成を指示するルール。文書を作成すべき部分は《 》で囲んでいます。

⑦ 適合例

実際のソースコードで、このルールに適合するように記述する場合の例を記載しています。

⑧ 不適合例

実際のソースコードで、このルールに違反する場合の例を記載しています。

⑨ 備考

C言語仕様上の注意、ルールの必要性、ルール違反時に生じる問題などを説明しています。

作法表中の用語

表内で使用している用語について説明します。

用語	説明
アクセス	変数の参照、及び変更を含む参照のこと。
型指定子	データの型を指定するもの。char、int、floatなどの基本的な型を指定するものと、プログラマが独自にtypedefで定義した型を指定するもの。
型修飾子	型に特定の性質を付け加えるもの。次の3つがある。 const、restrict、volatile
記憶クラス指定子	データが記憶される場所を指定するもの。次の4つがある。 auto、register、static、extern
境界調整	コンパイラがデータをメモリに配置するときの方法を示す。例えば、int型が2バイトの場合、必ずメモリの偶数アドレスから配置するようにし、奇数アドレスから配置しないようにすること。
トライグラフ	"??="、"??/"、"??{" のように決められた3文字をコンパイラが特定の1文字に解釈する文字表記のこと。 "??="、"??/"、"??{" はそれぞれ、"#"、"\", "{" に変換される。
生存期間	変数が生成されてから、プログラムからの参照が保証されている期間をいう。
多バイト文字	2バイト以上のデータで表現される文字。漢字、ひらがななどの全角文字や、Unicodeで表現される文字などがある。
ナルポインタ	いかなるデータ、関数へのポインタと比較しても等しくないポインタ。
ナル文字	文字列の最後を表現する文字。"\0"で表現される。
スコープ、有効範囲	変数名との識別子が使用可能であるプログラム上の範囲。 ファイルスコープは、スコープが、ファイルの終わりまでであること。
副作用	実行環境の状態に変化を起こす処理。次の処理が該当する。 volatileデータの参照や変更、データの変更、ファイルの変更、及びこれらの操作を行う関数呼出し。
ブロック	データ宣言、プログラムなどにおいて波括弧 "{"、"}" で囲んだ範囲をいう。
列挙型	enum型。いくつかの列挙されたメンバで構成される。
列挙子	列挙型 (enum型) のメンバのこと。

組込みソフトウェア向けコーディング作法

本パートでは、組込みソフトウェア向けのコーディング作法を掲載します。既に紹介したように、作法はソフトウェアに求められる品質特性（JIS X 25010）を参考に「信頼性」「保守性」「移植性」「効率性」の4つの特性の視点（品質概念）でカテゴライズされています。ただし、このカテゴライズは便宜上のものであり、いくつかの作法やルールについては、それを守ることによって信頼性と保守性の両方を向上するのに役立つものもあります。

また、このパートでは、それぞれの品質特性に関係するコーディング作法とその作法を実現するためのリファレンスルールを掲載しています。

信頼性 (Reliability)	R	作成したソフトウェアの信頼性を向上させるための作法を整理してあります。 主な視点は、 <ul style="list-style-type: none">・利用した際の不具合をできる限り少なくする・バグやインタフェース違反などに対する許容性 などを考慮しています。
保守性 (Maintainability)	M	修正や保守のしやすいソースコードを作成するための作法を整理してあります。 主な視点としては <ul style="list-style-type: none">・コードの理解しやすさ・修正のしやすさ・修正による影響の少なさ・修正したコードの確認のしやすさ などが含まれます。
移植性 (Portability)	P	ある環境下での動作を想定して作成したソフトウェアを他の環境に移植する場合に、できるだけ誤りなく効率的に移植できるようにするための作法を整理してあります。
効率性 (Efficiency)	E	作成したソフトウェアの性能やリソースを有効活用するための作法を整理してあります。 主な視点は、 <ul style="list-style-type: none">・処理時間を意識したコーディング・メモリサイズを考慮したコーディング などを考慮しています。

信頼性

組み込みソフトウェアの多くは製品に組み込まれて、我々の生活の中の様々なシーンで利用されています。このため、組み込みソフトウェアの中には極めて高い信頼性が求められるものも少なくありません。ソフトウェアの信頼性とは、ソフトウェアとしての誤った動作（障害の発生）をしないこと、誤動作をしてもソフトウェア全体やシステム全体の機能動作に影響を及ぼさないこと、誤動作が発生しても正常動作に速やかに復帰できることなどが求められます。

ソースコードレベルで、ソフトウェアの信頼性について気を付けるべきこととしては、このような誤動作を引き起こすような記述を極力避けるといった工夫が求められます。

- 信頼性 1 … 領域は初期化し、大きさに気を付けて使用する。
- 信頼性 2 … データは範囲、大きさ、内部表現に気を付けて使用する。
- 信頼性 3 … 動作が保証された書き方にする。

信頼性

1

領域は初期化し、 大きさに気を付けて使用する。

C言語を用いたプログラムでは、様々な変数が利用されます。こうした変数などについては、コンピュータ上で確保する領域を意識し、領域の初期化などを確実にしておかないと、思わぬ誤動作のもとになります。

また、C言語のポインタはポイントする先の領域を意識して利用しなければなりません。ポインタの使い方を誤るとシステム全体に重大な問題を引き起こす危険があるため、特に注意して利用する必要があります。

「信頼性 1」は、次の3つの作法で構成されます。

信頼性 1.1

領域は、初期化してから使用する。

信頼性 1.2

初期化は過不足ないことがわかるように記述する。

信頼性 1.3

ポインタの指す範囲に気を付ける。

領域は、初期化してから使用する。

R1.1.1

自動変数は宣言時に初期化する。または値を使用する直前に初期値を代入する。

選択指針

●

規約化

適合例

```
void func() {  
    int var1 = 0; /* 宣言時に初期化する */  
    int i;  
    var1++;  
    /* 使用する直前に初期値を代入 */  
    for (i = 0; i < 10; i++) {  
        ...  
    }  
}
```

不適合例

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

- 自動変数を初期化しないと、その値は不定となり、環境によって演算結果が異なる現象が発生する。初期化のタイミングは宣言時、または使用する直前とする。

R1.1.2

const 型変数は、宣言時に初期化する。

選択指針

●

規約化

適合例

```
const int N = 10;
```

不適合例

```
const int N;
```

- const 型変数は後から代入ができないので、宣言時に初期化すべきである。初期化しないと外部変数の場合は0、自動変数の場合は不定となるので、意図しない動作となる可能性がある。宣言時に未初期化でもコンパイラエラーにならないため、注意が必要である。

参考 C++ では const の未初期化はエラーとなる。

[関連ルール]

M1.11.1, M1.11.3

初期化は過不足ないことがわかるように記述する。

R1.2.1

要素数を指定した配列の初期化では、初期値の数は、指定した要素数と一致させる。

選択指針

●

規約化

適合例

```
char var[] = "abc";  
または  
char var[4] = "abc";
```

不適合例

```
char var[3] = "abc";
```

配列を文字列で初期化する際に、配列の大きさとしてナル文字分を確保せずとも宣言時にはエラーにならない。意図した記述であれば問題ないが、文字列操作関数などの引数として使用すると文字列の最後を示すナル文字がないため、意図しない動作となる可能性が高い。文字列の初期化の際には、最後のナル文字分まで確保する必要がある。

【関連ルール】

M2.1.1

R1.2.2

列挙型（enum 型）のメンバの初期化は、定数を全く指定しない、すべて指定する、または最初のメンバだけを指定する、のいずれかとする。

選択指針

規約化

適合例

```
/* E1 から E4 には異なる値が割り付けられる */  
enum etag { E1=9, E2, E3, E4 };  
enum etag var1;  
var1 = E3;  
/* var1 に入れた E3 と E4 が等しくなることはない */  
if (var1 == E4)
```

不適合例

```
/* 意図せずに E3 と E4 がどちらも 11 になる */  
enum etag { E1, E2=10, E3, E4=11 };  
enum etag var1;  
var1 = E3;  
/* E3 と E4 は等しいので、意図に反して真になる */  
if (var1 == E4)
```

列挙型のメンバに初期値を指定しない場合、直前のメンバの値に1を加えた値になる（最初のメンバの値は0）。初期値を指定したり、指定しなかったりすると、不用意に同じ値を割り当ててしまい、意図しない動作となる可能性がある。使い方にも依存するが、メンバの初期化は、定数を全く指定しない、すべて指定する、または最初のメンバだけを指定するのいずれかとし、同じ値が割り振られるのを防止した方がよい。

ポインタの指す範囲に気を付ける。

R1.3.1

- (1) ポインタへの整数の加減算 (++, --も含む) は使用せず、確保した領域への参照・代入は [] を用いる配列形式で行う。
- (2) ポインタへの整数の加減算 (++, --も含む) は、ポインタが配列を指している場合だけとし、結果は、配列の範囲内を指すようにする。

選択指針

●

規約化

選

適合例

```
#define N 10
int data[N];
int *p;
int i;
p = data;
i = 1;
(1)、(2) の適合例
data[i] = 10; /* OK */
data[i+3] = 20; /* OK */
(2) の適合例
*(p + 1) = 10;
```

不適合例

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

(1) の不適合例
*(p + 1) = 10; /* NG */
p += 2; /* NG */
(2) の不適合例
*(p + 20) = 10; /* NG */
```

ポインタに対する演算は、ポインタの指している先を分かりにくくする原因となる。すなわち、確保していない領域を参照したり、領域に書き込んだりするバグを埋め込む可能性が高くなる。領域の先頭を指している配列名を使った配列の添え字により、配列要素をアクセスする方が、安全なプログラムとなる。mallocなどによって獲得した動的メモリは配列と判断し、先頭ポインタを配列名と同等に扱う。

なお、多次元配列に対して、このルールは各部分配列に適用する。

(2) のルールにおいて、配列の最後の要素を1つ越えたところについては、配列要素にアクセスしない限り指してもよい。すなわち、int data[N]、p=dataとして、p+Nを、配列要素のアクセスに利用しない場合はルールに適合しており、*(p+N)のように配列要素のアクセスに利用する場合は不適合である。

R13.2

ポインタ同士の減算は、同じ配列の要素を指すポインタにだけ使用する。

選択指針	●
規約化	

適合例

```
ptrdiff_t off; /* ptrdiff_tは<stddef.h>にて
               定義されているポインタ減算結果
               の型 */

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

不適合例

```
ptrdiff_t off; /* ptrdiff_tは<stddef.h>にて
               定義されているポインタ減算結果
               の型 */

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

C言語では、ポインタ同士の減算を行った場合、各ポインタが指している要素の間に幾つ要素があるかが求まる。このとき、各ポインタが別の配列を指している、その間にどのような変数がレイアウトされるかは、コンパイラ依存であり、実行結果は保証されない。このようにポインタ同士の減算は、同じ配列内の要素を指している場合のみ意味がある。従って、ポインタ減算を行う場合には、同じ配列を指しているポインタ同士であることをプログラマが確認して行う必要がある。

【 関連ルール 】

R13.3

R13.3

ポインタ同士の大小比較は、同じ配列の要素、または同じ構造体のメンバを指すポインタにだけ使用する。

選択指針	●
規約化	

適合例

```
#define N 10
char var1[N];
void func(int i, int j) {
    if (&var1[i] < &var1[j]) {
        ...
    }
}
```

不適合例

```
#define N 10
char var1[N];
char var2[N];
void func(int i, int j) {
    if (&var1[i] < &var2[j]) {
        ...
    }
}
```

異なる変数のアドレス大小比較をしてもコンパイルエラーにならないが、変数の配置はコンパイラ依存なので意味のない大小比較となる。また、このような大小比較の動作は、定義されていない（未定義の動作）。

【 関連ルール 】

R13.2, R2.7.3

R13.4

restrict型修飾子は使用しない。

[MISRA C:2012 R8.14]

選択指針

規約化

適合例

不適合例

```
void f(int n, int * restrict p,  
int * restrict q) {  
    while (n-- > 0) {  
        *p++ = *q++;  
    }  
}  
  
void g(void) {  
    extern int d[100];  
  
    f(50, d+1, d); /* 未定義の動作 */  
}
```

restrict型修飾を行うことで、効率良いコードが生成できたり、コードチェッカなど静的解析の精度が上がる。しかし、対象となる領域がオーバーラップしないことをプログラマが保証しなければならず、コンパイラがエラーを出さないので危険性が伴う。

データは、範囲、大きさ、内部表現に気を付けて使用する。

プログラム内で扱う様々なデータは、その種類により内部的な表現が異なり、扱えるデータの範囲も異なります。こうした様々なデータを利用して演算などの処理を行なった場合、データを記述するときに、例えば、データの精度やデータの大きさなどに注意をしないと、思わぬ誤動作のもとになりかねません。このようにデータを扱う場合には、その範囲、大きさ、内部表現などを意識するように心がける必要があります。

信頼性 2.1

内部表現に依存しない比較を行う。

信頼性 2.2

論理値などが区間として定義されている場合、その中の一点（代表的な実装値）と等しいかどうかで判定を行ってはならない。

信頼性 2.3

データ型を揃えた演算や比較を行う。

信頼性 2.4

演算精度を考慮して記述する。

信頼性 2.5

情報損失の危険のある演算は使用しない。

信頼性 2.6

対象データが表現可能な型を使用する。

信頼性 2.7

ポインタの型に気を付ける。

信頼性 2.8

宣言、使用、定義に矛盾がないことをコンパイラがチェックできる書き方にする。

内部表現に依存しない比較を行う。

R2.1.1

浮動小数点式は、等価または非等価の比較をしない。

選択指針

●

規約化

適合例

```
#define LIMIT 1.0e-4
void func(double d1, double d2) {
    double diff = d1 - d2;
    if ((-LIMIT <= diff) && (diff <= LIMIT)) {
        ...
    }
}
```

不適合例

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

浮動小数点型は、ソースコード上に書かれた値と実装された値は完全に一致していないので、比較は許容誤差を考慮して判定する必要がある。

[関連ルール]

R2.1.2

R2.1.2

浮動小数点型変数はループカウンタとして使用しない。

選択指針

●

規約化

適合例

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

不適合例

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

浮動小数点型は、ループカウンタとして演算が繰り返されると、誤差が累積し、意図した結果が得られないことがある。このため、ループカウンタには整数型 (int 型) を使用すべきである。

[関連ルール]

R2.1.1

R2.1.3

構造体や共用体の比較に memcmp を使用しない。

選択指針

●

規約化

適合例

```
struct TAG {
    char c;
    long l;
};
struct TAG var1, var2;
void func() {
    if (var1.c == var2.c && var1.l == var2.l) {
        ...
    }
}
```

不適合例

```
struct TAG {
    char c;
    long l;
};
struct TAG var1, var2;
void func() {
    if (memcmp(&var1, &var2, sizeof(var1)) == 0)
    {
        ...
    }
}
```

構造体や共用体のメモリには、未使用の領域が含まれる可能性がある。その領域には何が入っているかわからないので、memcmp は使用すべきでない。比較する場合は、メンバ同士で比較する。

[関連ルール]

M1.6.2

信頼性

2.2

論理値などが区間として定義されている場合、その中の一点（代表的な実装値）と等しいかどうかで判定を行ってはならない。

R2.2.1

真偽を求める式の中で、真として定義した値と比較しない。

選択指針

規約化

適合例

```
#define FALSE 0
/* func1 は 0 と 1 以外を返す可能性がある */
void func2() {
    if (func1() != FALSE) {
        または
    if (func1()) {
        ...
    }
}
```

不適合例

```
#define TRUE 1
/* func1 は、0 と 1 以外を返す可能性がある */
void func2() {
    if (func1() == TRUE) {
        ...
    }
}
```

C 言語では、真は 0 ではない値で示され、1 とは限らない。

[関連ルール]

M1.5.2

データ型を揃えた演算や比較を行う。

R2.3.1

符号なし整数定数式は、結果の型で表現できる範囲内で記述する。

選択指針

規約化

適合例

```
#define MAX 0xffffUL /* long 型を指定する */
unsigned int i = MAX;
if (i < MAX + 1)
/*
    long が 32bit であれば、int の bit 数が
    違って問題ない */
```

不適合例

```
#define MAX 0xffffU
unsigned int i = MAX;
if (i < MAX + 1)
/*
    int が 16bit か 32bit かで結果が異なる。
    int が 16bit の場合、演算結果はラップアラウンド
    して比較結果は偽になる。int が 32 bit の場合、
    演算結果は int の範囲内に収まり、比較結果は
    真になる */
```

C言語の符号なし整数演算は、オーバーフローせずにラップアラウンドする（表現可能な最大数の剰余となる）。このため、演算結果が意図と異なっていることに気が付かない場合がある。例えば、同じ定数式でも、intのビット数が異なる環境では、演算結果がその型で表現できる範囲を超えた場合と超えない場合で結果が異なる。

R2.3.2

条件演算子(?:演算子)では、論理式は括弧で囲み、戻り値は2つとも同じ型にする。

選択指針

規約化

適合例

```
void func(int i1, int i2, long l1) {
    i1 = (i1 > 10) ? i2 : (int)l1;
```


不適合例

```
void func(int i1, int i2, long l1) {
    i1 = (i1 > 10) ? i2 : l1;
```

型が異なる記述を行った場合は、結果はどちらの型を期待しているかを明示するためにキャストする。

[関連ルール]

M1.4.1



ループカウンタとループ継続条件の比較に使用する変数は、同じ型にする。

選択指針	●
規約化	

適合例

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


不適合例

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

- ループの継続条件に、表現できる値の範囲が違う変数の比較を使用すると、意図した結果にならず、無限ループになる場合がある。

信頼性
2.4

演算精度を考慮して記述する。



演算の型と演算結果の代入先の型が異なる場合は、期待する演算精度の型へキャストしてから演算する。

選択指針	●
規約化	

適合例

```
int i1, i2;
long l;
double d;
void func() {
    d = (double)i1 / (double)i2; /* 浮動小数点型での除算 */
    l = ((long)i1) << i2; /* longでのシフト */
```

不適合例

```
int i1, i2;
long l;
double d;
void func() {
    d = i1 / i2; /* 整数型での除算 */
    l = i1 << i2; /* intでのシフト */
```

- 演算の型は演算に使用する式（オペランド）の型によって決まり、代入先の型は考慮されない。演算の型と代入先の型が異なる場合、誤って代入先の型での演算を期待していることがある。オペランドの型とは異なる型の演算を行いたい場合は、期待する型にキャストしてから演算する必要がある。

- [関連ルール]**
R2.5.1

R2.42

符号付きの式と符号なしの式の混在した算術演算、比較を行う場合は、期待する型に明示的にキャストする。

選択指針

規約化

適合例

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

不適合例

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

大小比較、乗除算など、演算を符号付きで行うか、符号なしで行うかによって結果が異なる演算もある。符号付き、符号なしを混在して記述した場合、どちらで行われるかは、それぞれのデータのサイズも考慮して決定されるため、常に符号なしで行われるとは限らない。このため、混在した算術演算を行う場合は、期待する演算が符号付きか符号なしかを確認し、期待した演算になるように明示的にキャストする必要がある。

注意 機械的にキャストするのではなく、使用するデータ型を変更した方がよい場合が多いので、まずデータ型の変更を検討する。

情報損失の危険のある演算は使用しない。

R2.5.1

情報損失を起こす可能性のあるデータ型への代入（＝演算、関数呼出しの実引数渡し、関数復帰）や演算を行う場合は、問題がないことを確認し、問題がないことを明示するためにキャストを記述する。

選択指針

○

規約化

適合例

```
/* 代入の例 */
short s;      /* 16ビット */
long l;       /* 32ビット */
void func() {
    s = (short)l;
    s = (short)(s + 1);
}
/* 演算の例 */
unsigned int var1, var2; /* intサイズが
                        16ビット */
var1 = 0x8000;
var2 = 0x8000;
if ((long)var1 + var2 > 0xffff) { /* 判定結果
                                は真 */
```

不適合例

```
/* 代入の例 */
short s;      /* 16ビット */
long l;       /* 32ビット */
void func() {
    s = l;
    s = s + 1;
}
/* 演算の例 */
unsigned int var1, var2; /* intサイズが
                        16ビット */
var1 = 0x8000;
var2 = 0x8000;
if (var1 + var2 > 0xffff) { /* 判定結果は偽
                            */
```

値をその型と異なる型の変数に代入すると、値が変わる（情報損失する）可能性がある。可能であれば代入先は同じ型とするのがよい。情報損失の恐れはない、または損失してもよいなど、意図的に異なる型へ代入する場合は、その意図を明示するためにキャストを記述する。

演算では、演算結果がその型で表現できる値の範囲を超えた場合、意図しない値になる可能性がある。安全のためには、演算結果がその型で表現できる値の範囲にあることを確認してから演算する。もしくは、より大きな値を扱える型に変換してから演算する。

注意 機械的にキャストするのではなく、使用するデータ型を変更した方がよい場合が多いので、まずデータ型の変更を検討する。

〔関連ルール〕

R2.4.1

R2.5.2

単項演算子 "-" は符号なしの式に使用しない。

選択指針

●

規約化

適合例

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

不適合例

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

- 符号なしの式に単項 '-' を使用することで、演算結果が元の符号なしの型で表現できる範囲外になった場合、予期しない動作となる可能性がある。
- 例えば、上記例で `if (-ui < 0)` と記述した場合、この `if` は真にはならない。

R2.5.3

unsigned char 型、または unsigned short 型のデータをビット反転 (~)、もしくは左シフト (<<) する場合、結果の型に明示的にキャストする。

選択指針

○

規約化

適合例

```
uc = 0x0f;  
if((unsigned char)(~uc) >= 0x0f)
```

不適合例

```
uc = 0x0f;  
if((~uc) >= 0x0f) /* 真にならない */
```

- unsigned char または unsigned short の演算結果は signed int となる。演算によって符号ビットがオンになると、期待した演算結果にならない場合がある。このため、期待する演算の型へのキャストを明示する。不適合例では、`~uc` は負の値となるので必ず偽となる。

[関連ルール]

R2.5.4



シフト演算子の右辺の項はゼロ以上、左辺の項のビット幅未満でなければならない。

選択指針	●
規約化	

適合例

```
unsigned char  a; /* 8ビット */
unsigned short b; /* 16ビット */
b = (unsigned short)a << 12; /* 16ビットとして
                               処理していることが明示的 */
```

不適合例

```
unsigned char  a; /* 8ビット */
unsigned short b; /* 16ビット */
b = a << 12; /* シフト数に誤りの可能性あり */
```

シフト演算子の右辺（シフト数）の指定が、負の値の場合と左辺（シフトされる値）のビット幅（int よりサイズが小さい場合はintのビット幅）以上の場合の動作は、C言語規格で定義されておらず、コンパイラによって異なる。

左辺（シフトされる値）がintより小さいサイズの型の場合に、シフト数としてintのビット幅までの値を指定することは、言語規格で動作が定義されているが意図が分かりにくい。

【 関連ルール 】

R2.5.3

対象データが表現可能な型を使用する。



- (1) ビットフィールドに使用する型は signed int と unsigned int だけとし、1 ビット幅のビットフィールドが必要な場合は signed int 型でなく、unsigned int 型を使用する。
- (2) ビットフィールドに使用する型は signed int、unsigned int、または _Bool とし、1 ビット幅のビットフィールドが必要な場合は、unsigned int 型または _Bool 型を使用する。
- (3) ビットフィールドに使用する型は signed int、unsigned int、_Bool、または処理系が許容している型のうち signed と unsigned を指定した型または enum 型を使用する。1 ビット幅のビットフィールドが必要な場合は、unsigned を指定した型、または _Bool 型を使用する。

選択指針	
規約化	選

適合例

(1) の適合例

```
struct S {
    signed int  m1:2;
    unsigned int m2:1;
    unsigned int m3:4;
};
```

(2) の適合例

```
struct S {
    _Bool      m1:1;
};
```

(3) の適合例

```
struct S {
    unsigned char m1:2; /* 処理系定義で許可されている場合 OK */
    char          m2:1; /* 処理系定義で許可されている場合 OK */
    enum E        m3:2; /* 処理系定義で許可されている場合 OK */
    enum          m4:4; /* 処理系定義で許可されている場合 OK */
};
```

不適合例

(1) の不適合例

```
struct S {
    int      m1:2; /* NG: 符号指定がない
    (2)、(3) にも不適合 */
    signed int m2:1; /* NG: ビット幅1の
    signed int 型の使用 (2)、(3) にも不適合 */
    unsigned char m3:4; /* NG: char 型の使用
    (2) にも不適合 char が処理系定義で許されている
    場合、(3) には OK */
    enum E      m4:2; /* NG: enum 型の使用
    (2) にも不適合 enum が処理系定義で許されている
    場合、(3) には OK */
    _Bool      m5:1; /* NG: _Bool 型の使用
    (2)、(3) には OK */
};
```

(2) の不適合例

```
struct S {
    int      m1:2; /* NG: 符号指定がない
    (1)、(3) にも不適合 */
    signed int m2:1; /* NG: ビット幅1の signed
    int 型の使用 (1)、(3) にも不適合 */
    unsigned char m3:4; /* NG: char 型の使用 (1) にも不適合
    char が処理系定義で許されている場合、(3) には適合 */
    enum E      m4:2; /* NG: enum 型の使用
    (1) にも不適合 enum が処理系定義で許されている場合
    (3) には OK */
};
```

(3) の不適合例

```
struct S {
    int      m1:2; /* NG: 符号指定がない
    (1)、(2) にも不適合 */
    signed int m2:1; /* NG: ビット幅1の
    signed int 型の使用 (1)、(2) にも不適合 */
};
```

- (1) C90との互換性のために、C90で規定されている「signedまたはunsignedの符号指定のあるint型」のみを使用し、コンパイラによって符号付き符号なしのいずれかが異なる「符号指定のないint型」を使用しない。
 - (2) C99で規定されている「signed またはunsignedの符号指定のあるint型」または_Bool型のみを使用し、コンパイラによって符号付き符号なしのいずれかが異なる「符号指定のないint型」を使用しない。
 - (3) C99の言語規格で定められている型に加えて処理系定義の型を使用可能とする。ただし、コンパイラによって符号付き符号なしのいずれかが異なる「符号指定のない整数型」を使用しない。いずれのルールも符号指定のない整数型の使用を禁止している。
- 符号指定のないintをビットフィールドに使用した場合、符号付き、符号なしのどちらで使われるかはコンパイラによって異なる。そのため、符号指定のないint型はビットフィールドには使用しない。また、1ビットの符号付き整数型で表現できる値は-1と0のみとなるので、1ビットの整数型ビットフィールドにはunsignedを指定する。

R2.6.2

ビット列として使用するデータは、符号付き型ではなく、符号なし型で定義する。

選択指針

規約化

適合例

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

不適合例

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

- 符号付き型に対するビット単位の演算（～、<<、>>、&、^、|）の結果は、コンパイラによって異なる可能性がある。

ポインタの型に気を付ける。



- (1) ポインタ型は、他のポインタ型及び整数型と相互に変換してはならない。ただし、データへのポインタ型における void* 型との相互変換は除く。
- (2) ポインタ型は、他のポインタ型、及びポインタ型のデータ幅未満の整数型と相互に変換してはならない。ただし、データへのポインタ型における void* 型との相互変換は除く。
- (3) ポインタ型は、他のポインタ型、及びポインタ型のデータ幅未満の整数型と相互に変換してはならない。ただし、データへのポインタ型における、他のデータへのポインタ型及び void* 型との相互変換は除く。

選択指針	○
規約化	選

適合例

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

(1) の適合例
ip = (int*)vp;

(2) の適合例
i=(intptr_t)ip;

(3) の適合例
i=(intptr_t)fp;
cp = (char*)ip;

不適合例

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

(1) の不適合例
ip = (int*)cp;

(2) の不適合例
c =(char) ip;

(3) の不適合例
ip =(int*) fp;

ポインタ型の変数を他のポインタ型にキャストや代入をすると、ポインタの指す先の領域がどのようなデータなのか分かりにくくなる。CPUによっては、ワード境界を指さないポインタを使ってポインタの指す先を int 型でアクセスすると、実行時エラーが発生するものもあり、ポインタの型を変更すると思わぬバグになる危険がある。ポインタ型の変数は、他のポインタ型にキャストや代入をしない方が安全である。ポインタ型を整数型に変換することも前述の問題と同じ危険性があり、必要な場合は、経験者を交えたレビューを行う。さらに、int 型の扱う値の範囲とポインタ型の扱う値の範囲に対する注意も必要である。int 型サイズが32ビットにも関わらず、ポインタ型サイズが64ビットということもあるので、事前にコンパイラの仕様を確認しておく。

<stdint.h>では intptr_t 及び uintptr_t という型が定義されることが規定されており、それぞれポインタ型を格納可能なデータ幅をもつ符号付き整数と符号なし整数を表す。ポインタ型と整数型の間で変換を行う際には、これらの型を利用するとよい。

☆ポインタ変換に関するルール

R.2.7.1のルールで解説した通り、ポインタ型の変数を他のポインタ型へ変換（代入）するのは意図した通りにならない可能性があり、むやみに記述するのは危険である。ポインタ変換に関するルールを以下の表にまとめた。

表の縦列が変換（代入）元、表の横列が（代入）先となる。○は変換してよいこと、×は変換してはいけないことを示す。

項目 (1)

		変換先			
		データへのポインタ	関数へのポインタ	void へのポインタ	その他の型
変換元	データへのポインタ	×	×	○	×
	関数へのポインタ	×	×	×	×
	void へのポインタ	○	×	—	×
	その他の型	×	×	×	

項目 (2)

		変換先				
		データへのポインタ	関数へのポインタ	void へのポインタ	整数型	
					サイズがポインタ型のデータ幅未満	サイズがポインタ型のデータ幅以上
変換元	データへのポインタ	×	×	○	×	○
	関数へのポインタ	×	×	×	×	○
	void へのポインタ	○	×	—	×	○
	整数型	サイズがポインタ型のデータ幅未満	×	×		
		サイズがポインタ型のデータ幅以上	○	○		

項目 (3)

		変換先				
		データへのポインタ	関数へのポインタ	void へのポインタ	サイズがポインタ型のデータ幅未満	サイズがポインタ型のデータ幅以上
変換元	データへのポインタ	○	×	○	×	○
	関数へのポインタ	×	×	×	×	○
	void へのポインタ	○	×	—	×	○
	整数型	サイズがポインタ型のデータ幅未満	×	×		
		サイズがポインタ型のデータ幅以上	○	○		

R2.72

ポインタで指し示された型から `const` 修飾や `volatile` 修飾を取り除くキャストを行ってはならない。[MISRA C:2012 R11.8]

選択指針

○

規約化

適合例

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

不適合例

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

`const` や `volatile` 修飾された領域は、参照しかされない領域であったり、最適化をしてはならない領域なので、その領域に対するアクセスに注意しなければならない。これらの領域を指すポインタに対し、`const` や `volatile` を取り除くキャストを行ってしまうと前述の注意項目が見えなくなり、コンパイラはプログラムの誤った記述に対し、何もチェックできなくなったり、意図しない最適化を行ってしまったりする可能性がある。

R2.73

ポインタが負かどうかの比較をしない。

選択指針

●

規約化

適合例

—

不適合例

```
int * func1() {  
    ...  
    return (int*)-1;  
}  
int func2() {  
    ...  
    if (func1() < 0) { /* 負かどうかの比較のつもり */  
        ...  
    }  
    return 0;  
}
```

ポインタと0との大小比較は意味がないので注意が必要である。
0は、比較の対象がポインタの場合、コンパイラによってナルポインタに変換される。従って、それはポインタ同士の比較であり、期待する動作にならない可能性がある。

[関連ルール]

R1.3.3

信頼性
2.8

宣言、使用、定義に矛盾がないことをコンパイラが
チェックできる書き方にする。



引数をもたない関数は、引数の型を void として宣
言する。

選択指針

○

規約化

適合例

```
int func(void) ;
```

不適合例

```
int func();
```

int func(); は、引数がない関数の宣言ではなく、旧式 (K&R 形式) の宣言で引数の数と型が不明という
意味である。引数がない関数を宣言する場合は void を明記する。

〔 関連ルール 〕

R2.8.3



(1) 可変個引数をもつ関数を定義してはならない。

[MISRA C:2004 16.1]

(2) 可変個引数をもつ関数を使用する場合は、《処理系での
動作を文書化し、使用する》。

選択指針

規約化

選文

適合例

(1) の適合例

```
int func(int a, char b);
```

不適合例

(1) の不適合例

```
int func(int a, char b, ... );
```

可変個引数関数が、使用する処理系でどのような動作をするかを理解した上で使用しないと期待する動
作をしない、スタックオーバーフローを発生するなどの可能性がある。

また、引数を可変とした場合、引数の個数と型が明確に定義されないので、可読性が低下する。

MISRA C:2012 では、<stdarg.h> にて定義される関数の使用を禁じている。

〔 関連ルール 〕

R2.8.3

R2.8.3

1つのプロトタイプ宣言は1箇所に記述し、それが関数呼出し及び関数定義の両方から参照されるようにする。

選択指針

○

規約化

適合例

```
-- file1.h --
void f(int i);

-- file1.c --
#include "file1.h"
void f(int i) { ... }

-- file2.c --
#include "file1.h"
void g(void) { f(10); }
```

不適合例

```
-- file1.c --
void f(int i);      /* それぞれのファイルで宣言 */
void f(int i) { ... }

-- file2.c --
void f(int i);      /* それぞれのファイルで宣言 */
void g(void) { f(10); }
```

本ルールはプロトタイプ宣言と関数定義の不整合を防止するためのルールである。

【関連ルール】

R2.8.1, R2.8.2

動作が保証された書き方にする。

プログラムの仕様上、あり得ないケースについても、想定外の事象が起こることを考慮し、エラー処理を漏れなく記述することも必要となります。また、演算子の優先順位付など、言語仕様に頼らない書き方も安全性を高めます。高い信頼性を実現させるためには、誤動作につながる記述を極力避け、できるだけ動作が保証された安全な書き方をすることが望めます。

信頼性 3.1

領域の大きさを意識した書き方にする。

信頼性 3.2

実行時にエラーになる可能性のある演算に対しては、エラーケースを迂回させる。

信頼性 3.3

関数呼出しではインタフェースの制約をチェックする。

信頼性 3.4

再帰呼出しは行わない。

信頼性 3.5

分岐の条件に気を付け、所定の条件以外が発生した場合の処理を記述する。

信頼性 3.6

評価順序に気を付ける。

領域の大きさを意識した書き方にする。

R3.1.1

- (1) 配列のextern宣言の要素数は必ず指定する。
- (2) 要素数が省略された初期化付き配列定義に対応した配列のextern宣言を除き配列のextern宣言の要素数は必ず指定する。

選択指針	○
規約化	選

適合例

```
(1) の適合例
extern char *mes[3];
...
char *mes[] = {"abc", "def", NULL};

(2) の適合例
extern char *mes[];
...
char *mes[] = {"abc", "def", NULL};

(1)、(2) の適合例
extern int var1[MAX];
...
int var1[MAX];
```

不適合例

```
(1) の不適合例
extern char *mes[];
...
char *mes[] = {"abc", "def", NULL};

(1)、(2) の不適合例
extern int var1[];
...
int var1[MAX];
```

配列の大きさを省略してextern宣言しても、エラーにはならない。しかし、大きさが省略されていると、配列の範囲外のチェックに支障が生じる場合がある。このため、配列の大きさは明示して宣言した方がよい。ただし、初期値の個数で配列の大きさを決定し、宣言時に大きさが決まらない場合などは、宣言時の配列の大きさを省略した方がよい場合もある。

【関連ルール】

R3.1.2

R3.1.2

配列を順次にアクセスするループの継続条件には、配列の範囲内であるかの判定を入れる。

選択指針	
規約化	

適合例

```
char var1[MAX];
for (i = 0; i < MAX && var1[i] != 0; i++) {
    /* var1 配列に 0 が未設定の場合でも、配列の範囲外
    アクセスの危険なし */
    ...
}
```

不適合例

```
char var1[MAX];
for (i = 0; var1[i] != 0; i++) { /*
    var1 配列に 0 が未設定の場合、配列の範囲外アクセス
    の危険あり */
    ...
}
```

領域外のアクセスを防ぐためのルールである。

[関連ルール]
R3.1.1

R3.1.3

指示付きの初期化子で初期化する配列のサイズは明示する。

選択指針	
規約化	

適合例

```
int a[ 5 ] = { [ 0 ] = 1 };
```

不適合例

```
int b[ ] = { [ 0 ] = 1 };
```

定義時に配列のサイズが明示されないと、初期化される要素の中で最大のインデックスとして決定される。指示付きの初期化子を用いると、どれが最大のインデックスとなる初期化であるかが明確でないこともある。

[関連ルール]
R3.1.1

R3.1.4

可変長配列型は使用しない。 [MISRA C:2012 R18.8]

選択指針

規約化

適合例

```
#define MAX 1024
void func(void) {
    int a[MAX]; /* OK 最大長の配列を確保 */
```

不適合例

```
void func(int n) {
    int a[n]; /* NG 可変長配列 */
```

可変長配列の利用は以下の問題がある。

- スタックオーバーフローの危険

可変長配列はスタック領域に割り付けられる。このため、可変長配列サイズが大きいとスタックオーバーフローとなる危険がある。

- C 言語規格で定義されていない動作

可変長配列のサイズが正の値でない場合の振る舞いは、C 言語規格で定義されていない。

- 配列サイズの勘違い

```
int y = 10;
typedef int INTARRAY[y];
y = 20;
INTARRAY z; /* z の配列サイズは 20 ではなく 10 */
```

信頼性
3.2

実行時にエラーになる可能性のある演算に対しては、エラーケースを迂回させる。



R3.2.1

除算や剰余算の右辺式は、0でないことを確認してから演算を行う。

選択指針

規約化

適合例

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

不適合例

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

明らかに0でない場合を除き、除算や剰余算の右辺が0でないことを確認してから演算する。そうしない場合、実行時に0除算のエラーが発生する可能性がある。

【関連ルール】

R3.2.2, R3.3.1



R3.2.2

ポインタは、ナルポインタでないことを確認してからポインタの指す先を参照する。

選択指針

規約化

適合例

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

不適合例

```
*p = 1;
```

【関連ルール】

R3.2.1, R3.3.1

関数呼出しではインターフェースの制約をチェックする。

R3.3.1

関数がエラー情報を戻す場合、エラー情報をテストしなければならない。[MISRA C:2012 D4.7]

選択指針

規約化

適合例

```
p = malloc(BUFSIZE);
if (p == NULL)
    /* 異常処理 */
else
    *p = '\0';
```

不適合例

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

関数に返却値がある場合にそれを利用していないコードはエラーの可能性はある。参照が不要なケースはvoidへキャストするなど、不要なことを明示するルールをプロジェクトで決めることも考えられる。

【関連ルール】

R3.2.1, R3.2.2, R3.5.1, R3.5.2



関数は、処理の開始前に引数の制約をチェックする。

選択指針	
規約化	

適合例

```
int func(int para) {
    if (!(MIN <= para) && (para <= MAX))
        return range_error;
    /* 通常処理 */
    ...
}
```

不適合例

```
int func(int para) {
    /* 通常処理 */
    ...
}
```

引数の制約を呼び出し側でチェックするか、呼び出される側でチェックするかは、インターフェースの設計の問題である。しかし同じチェックを1箇所にとまとめ、チェック忘れを防止するためには、呼び出される関数側でチェックする。

ライブラリなどで、呼び出される関数を変更できない場合には、ラップ関数を作成する。

ラップ関数の例：

```
int func_with_check(int arg) {
    /* arg が引数制約に違反している場合 range_error を返す */
    /* そうでない場合 func を呼び出して結果を返す */
}
/* ラップ関数を利用した呼出し
if (func_with_check(para) == range_error) {
    /* エラー処理 */
}
```

なお、C99では仮引数の配列宣言においてstatic修飾子を用いることで配列要素数の下限を指定することができる。例えば、以下の関数宣言では、配列aの実引数に関して、その要素数の下限を3と指定している。

```
void func(int a[static 3]);
```

このような指定を行うことにより、コンパイラなどのツールが実引数をチェックすることを期待できる。

再帰呼出しは行わない。



関数は、直接的か間接的に関わらず、その関数自身を呼び出してはならない。[MISRA C:2012 R17.2]

選択指針

規約化

適合例

—

不適合例

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

- 再起呼出しは実行時の利用スタックサイズが予測できないためスタックオーバーフローの危険がある。

分岐の条件に気を付け、所定の条件以外が発生した場合の処理を記述する。

R3.5.1

if-else if 文は、最後に else 節を置く。

通常、else 条件が発生しないことが分かっている場合は、次のいずれかの記述とする。

《(1) else 節には、例外発生時の処理を記述する。

(2) else 節には、プロジェクトで規定したコメントを入れる。》

選択指針

○

規約化

規

適合例

```
/* else条件が通常発生しない場合の
   if-else if文のelse節 */
if (var1 == 0) {
    ...
} else if (0 < var1) {
    ...
} else {
    /* 例外処理を記述する */
    ...
}
...
if (var1 == 0) {
    ...
} else if (0 < var1) {
    ...
} else {
    /* NOT REACHED */
}
```

不適合例

```
/* else節のないif-else if文 */
if (var1 == 0) {
    ...
} else if (0 < var1) {
    ...
}
```

else 節がないと、else 節を書き忘れているのか、else 節が発生しない if-else if 文なのかが分からなくなる。通常、else 条件が発生しないことが分かっている場合でも次のように else 節を書くことによって想定外の条件が発生した場合のプログラムの動作を予測することができる。

- ・ else 条件に想定外の条件に対する動作を記述する（万が一、else 条件が発生した場合のプログラムの動作を決めておく）。

また、else 条件が発生しないコメントを記述するだけでも、プログラムが分かりやすくなる。

- ・ /* NOT REACHED */ のように、プロジェクトで規定した else 条件が発生しないことを明示するコメントを記述し、else 節の書き漏れではないことを表現する。

【 関連ルール 】

R3.3.1, R3.5.2

R3.5.2

switch文は、最後にdefault節を置く。
通常、default条件が発生しないことが分かっている場合は、次のいずれかの記述とする。
《(1) default節には、例外発生時の処理を記述する。
(2) default節には、プロジェクトで規定したコメントを入れる。》

選択指針

○

規約化

規

適合例

```
/* default条件が通常発生しないswitch文の
default節 */
switch(var1) {
case 0:
    ...
    break;
case 1:
    ...
    break;
default:
    /* 例外処理を記述する */
    ...
    break;
}
...
switch(var1) {
case 0:
    ...
    break;
case 1:
    ...
    break;
default:
    /* NOT REACHED */
    break;
}
```

不適合例

```
/* default節のないswitch文 */
switch(var1) {
case 0:
    ...
    break;
case 1:
    ...
    break;
}
```

default節がないと、default節を書き忘れているのか、default節が発生しないswitch文なのかが分からなくなる。

通常、default条件が発生しないことが分かっている場合でも、次のようにdefault節を書くことによって想定外の条件が発生した場合のプログラムの動作を予測することができる。

- ・ default条件に想定外の条件に対する動作を記述する（万が一、default条件が発生した場合のプログラムの動作を決めておく）。

また、default条件が発生しないコメントを記述するだけでも、プログラムが分かりやすくなる。

- ・ /* NOT REACHED */のように、プロジェクトで規定したdefault条件が発生しないことを明示するコメントを記述し、default節の書き漏れではないことを表現する。

- [関連ルール]
- R3.3.1, R3.5.1
- M3.1.4



ループカウンタの比較に等価演算子 (==、!=) は使用しない。

選択指針	
規約化	

適合例

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

不適合例

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

- ループカウンタの変化量が1でない場合、無限ループになる可能性があるので、ループ回数を判定する
- 比較では、等価演算子 (==、!=) は使用しない (<=、>=、<、>を使用する)。

R3.6.1

変数の値を変更する記述をした同じ式内で、その変数を参照、変更しない。

選択指針

●

規約化

適合例

```
f (x, x);  
x++;  
または  
f (x + 1, x);  
x++;
```

不適合例

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

複数の引数を持つ関数の各実引数の実行（評価）の順序は、コンパイラは保証していない。引数は右から実行されたり、左から実行されたりする。また、+演算のような2項演算の左式と右式の実行の順序も、コンパイラは保証していない。このため、引数並びや2項演算式内で、1つのオブジェクトの更新と参照を行うと、その実行結果が保証されない。実行結果が保証されないこのような問題を副作用問題と呼んでいる。副作用問題が発生する記述はしてはならない。

このルールでは、副作用問題の発生しない次のような記述については禁止していない。

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

〔 関連ルール 〕

R3.6.2

M1.8.1

R3.6.2

実引数並び、及び2項演算式に、副作用をもつ関数呼出し、volatile 変数を、複数記述しない。

選択指針

○

規約化

適合例

```
1.
extern int G_a;
x = func1();
x += func2();
...
int func1(void) {
    G_a += 10;
    ...
}
int func2(void) {
    G_a -= 10;
    ...
}

2.
volatile int v;
y = v;
f(y, v);
```

不適合例

```
1.
extern int G_a;
x = func1() + func2(); /* 副作用問題あり */
...
int func1(void) {
    G_a += 10;
    ...
}
int func2(void) {
    G_a -= 10;
    ...
}

2.
volatile int v;
f(v, v);
```

複数の引数をもつ関数の各実引数の実行（評価）の順序は、コンパイラは保証していない。引数は右から実行されたり、左から実行されたりする。また、+演算のような2項演算の左式と右式の実行の順序も、コンパイラは保証していない。このため、引数並びや2項演算式内で、副作用をもつ関数呼出しやvolatile変数を複数記述すると、その実行結果が保証されない場合がある。このような危険な記述は避けるべきである。

【関連ルール】

R3.6.1

M1.8.1

R3.6.3

sizeof 演算子は、副作用がある式に用いてはならない。

選択指針

●

規約化

適合例

```
x = sizeof(i);  
i++;  
y = sizeof(int[i]);  
i++;
```

不適合例

```
x = sizeof(i++);  
y = sizeof(int[i++]);
```

sizeof 演算子の括弧の中の式は、C90 までは式の型のサイズが求められるだけで、式の実行は行われなかった。

このため、sizeof(i++) のように ++ 演算子を記述しても、i はインクリメントされない。一方、C99 では可変長配列型の場合に式の評価が行われる場合があり、i は sizeof(int[i++]) では、++ 演算子で i がインクリメントされる。このような記述は誤解をまねきやすいため行わないほうがよい。

保守性

多くの組み込みソフトウェア開発では、ひとたび作成したソフトウェアに手を加えるといった保守作業も必要になります。

保守の原因は様々ですが、例えば、

- ・リリースしたソフトウェアの一部に不具合などが見つかリ修正をする場合
- ・製品に対する市場からの要求などに応じて、既存ソフトウェアをベースに、新たな機能を追加する場合

などが考えられます。

このように作成したソフトウェアに何らかの手を加える場合、その作業をできるだけ誤りなく効率的に行えるかどうか重要な特質になります。システムの世界では、これを保守性と呼びます。

ここでは、組み込みソフトウェアのソースコードに関して、保守性を維持し、向上させるための作法を整理してあります。

- 保守性 1 … 他人が読むことを意識する。
- 保守性 2 … 修正誤りのないような書き方にする。
- 保守性 3 … プログラムはシンプルに書く。
- 保守性 4 … 統一した書き方にする。
- 保守性 5 … 試験しやすい書き方にする。

保守性

1

他人が読むことを意識する。

ソースコードは、実際に作成した技術者以外の技術者が再利用したり、保守したりといった場合も十分に考えられます。このためソースコードは、将来第三者が読むことを考慮して、分かりやすい表現にしておくことが必要になります。

保守性 1.1

使用しない記述を残さない。

保守性 1.2

紛らわしい書き方をしない。

保守性 1.3

特殊な書き方はしない。

保守性 1.4

演算の実行順序が分かりやすいように記述する。

保守性 1.5

省略すると誤解をまねきやすい演算は、明示的に記述する。

保守性 1.6

領域は 1 つの利用目的に使用する。

保守性 1.7

名前を再使用しない。

保守性 1.8

勘違いしやすい言語仕様を使用しない。

保守性 1.9

特殊な書き方は意図を明示する。

保守性 1.10

マジックナンバーを埋め込まない。

保守性 1.11

領域の属性は明示する。

保守性 1.12

コンパイルされない文でも正しい記述を行う。

使用しない記述を残さない。

M1.11

使用しない関数、変数、引数、typedef、タグ、ラベル、マクロなどは宣言（定義）しない。

選択指針

○

規約化

適合例

```
void func(void) {  
  
    コールバック関数が必要な場合  
    int cbfunc1(int arg1, int arg2);  
    int cbfunc2(int arg1, int);  
    /* コールバック関数の型がint (*)(int, int)と  
       決まっている場合、第2引数は利用しなくても必要 */
```

不適合例

```
void func(int arg) {  
    /* arg未使用 */
```

使用しない関数、変数、引数、ラベルなどの宣言（定義）は、削除し忘れたのか、記述を誤っているかの判断が難しいため、保守性を損なう。

ただしコールバック関数の場合は、関数の型を揃えるために引数の名前を記述しないことで引数を使用していないことを明示する。

〔関連ルール〕

M1.9.1, M4.7.2

M1.12

コードの一部を"コメントアウト"すべきでない。

[MISRA C:2012 D4.4]

選択指針

○

規約化

適合例

```
...  
#if 0    /* ~のため、無効化 */  
    a++;  
#endif  
...
```

不適合例

```
...  
/* a++; */  
...
```

無効としたコード部を残すことは、コードを読みにくくするため本来避けるべきである。
ただし、コード部の無効化が必要な場合は、コメントアウトせず #if 0 で囲むなど、無効化したコード部を明示するルールを決めておく。

【関連ルール】

M1.12.1, M4.7.2

保守性 1.2

紛らわしい書き方をしない。

M1.2.1

- (1) 1つの宣言文で宣言する変数は1つとする（複数宣言しない）。
- (2) 同じような目的で使用する同じ型の自動変数は、1つの宣言文で複数宣言してもよいが、初期化する変数と初期化をしない変数を混在させてはならない。

選択指針

規約化

選

適合例

```
(1) の適合例
int i;
int j;
(2) の適合例
int i, j;
int k = 0;

int *p;
int i;
```

不適合例

```
(1) の不適合例
int i, j;
(2) の不適合例
int i, j, k = 0; /* 初期化のあるもの
                 ないものが混在 (NG) */

int *p, i; /* 型の異なる変数が混在 (NG) */
```

int *p; と宣言した場合、型は int * であるが、int *p, q; と宣言した場合、q の型は int * ではなく、int となる。

【関連ルール】

M1.6.1

M1.22

適切な型を示す接尾語が使用できる定数記述には、接尾語をつけて記述する。long 型整数定数を示す接尾語は大文字の "L" のみ使用する。

選択指針

規約化

適合例

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

f = f + 1.0F; /* floatの演算であることを
             明示する */
func(1L); /* Lは大文字で記述する */
if (ui < 0x8000U) { /* unsignedの比較であることを
                  明示する */
    ...
}
```

不適合例

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

f = f + 1.0;
func(1l); /* 1l (エル) は 11 と紛らわしい */
if (ui < 0x8000) {
    ...
}
```

基本的に接尾語がない場合は、整数はint型、浮動小数点定数はdouble型となる。ただし、整数でint型で表現できない値を記述した場合はその値を表現できる型になる。このため、0x8000はintが16bitの場合はunsigned intであるが、intが32bitの場合はsigned intとなる。unsignedとして使用したい場合は、接尾語として "U" を明記する必要がある。また、浮動小数点数のfloat型とdouble型の演算速度が異なるターゲットシステムの場合、float型の変数と浮動小数点定数の演算を行う際に、浮動小数点定数に接尾語 "F" がないと、その演算はdouble型の演算になるので注意が必要である。浮動小数点定数は、小数点の左右に少なくともひとつの数字を記述するなど、浮動小数点定数であることを見分けやすくする工夫が必要である。

【関連ルール】

M1.8.5

M1.23

長い文字列リテラルを表現する場合には、文字列リテラル内で改行を使用せず、連続した文字列リテラルの連結を使用する。

選択指針

規約化

適合例

```
char abc[] = "aaaaaaaa¥n"
             "bbbbbbbb¥n"
             "cccccc¥n";
```

不適合例

```
char abc[] = "aaaaaaaa¥n¥
             bbbbbbbb¥n¥
             cccccc¥n";
```

複数行にわたる長い文字列を表現したい場合、複数の文字列リテラルを連結した書きの方が見やすい。

M1.31

switch(式)の式には、真偽結果を求める式を記述しない。

選択指針

●

規約化

適合例

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

不適合例

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

- 真偽結果を求める式をswitch文に使用すると、分岐数は2つになり、多分岐命令であるswitch文を使用する必要性は低くなる。switch文は、default節の誤記や、break文の記述漏れなど、if文と比較して間違いが発生する可能性が高いので、3分岐以上にならない場合は、if文を使用することを推奨する。

M1.32

switch文のcaseラベル及びdefaultラベルは、switch文本体の複文（その中に入れ子になった複文は除く）にのみ記述する。

選択指針

規約化

適合例

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

不適合例

```
switch (x) { /* switch文本体の複文 */
case 1:
{ /* 入れ子になった複文 */
    ...
}
case 2: /* 入れ子になった複文にcaseラベルを記述しない */
    ...
    break;
default:
    ...
    break;
}
```

M1.33

関数や変数の定義や宣言では型を明示的に記述する。

選択指針

規約化

適合例

```
extern int global;

int func(void) {
    ...
}
```

不適合例

```
extern global;

func(void) {
    ...
}
```

関数や変数の定義や宣言でデータ型を記述しない場合 int 型と解釈されるが、明示的にデータ型を記述した方が見やすくなる。

C99 では、このようなデータ型を明示しない記述は言語規格で禁止されており、コンパイラがエラーとして検出する。

〔関連ルール〕

M4.5.1

保守性 1.4

演算の実行順序が分かりやすいように記述する。

M1.41

&& 演算や || 演算の右式と左式は二項演算を含まない式か () で囲まれた式を記述する。ただし、&& 演算が連続して結合している場合や、|| 演算が連続して結合している場合は、&& 式や || 式を () で囲む必要はない。

選択指針

規約化

適合例

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

不適合例

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

&& や || の各項は、優先順位が紛らわしくないような式にするというのが本ルールである。単項や後置、キャスト以外の演算子が含まれる式に対し () で囲むことにより、&& や || の各項の演算を目立たせ、可読性を向上させることが目的である。なお、! 演算については、初心者には優先順位が紛らわしいため、() で囲むというルールにすることも考えられる。

〔関連ルール〕

R2.3.2

M1.5.2

M1.4.2

《演算の優先順位を明示するための括弧の付け方を規定する。》

選択指針

規約化

規

適合例

```
a = (b << 1) + c;
または
a = b << (1 + c);
```

不適合例

```
a = b << 1 + c; /* 優先順位間違っている
可能性あり */
```

C言語の演算子の優先順位は見間違いやすいため、例えば以下のようなルールを決めるとよい。
式中に優先順位の異なる複数の2項演算子を含む場合には優先順位を明示するための括弧を付ける。ただし、四則演算に関しては括弧を省略してもよい。
演算子の優先順位とその解釈については、MISRA C:2012 Rule 12.1 (p.103) 参照。

〔関連ルール〕

M1.5.1

保守性 1.5

省略すると誤解をまねきやすい演算は、明示的に記述する。

M1.5.1

関数識別子（関数名）には、前に&を付けるか、括弧付の仮引数リスト（空でも可）を指定して使用しなければならない。[MISRA C:2004 16.9]

選択指針

規約化

適合例

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

if (func()) {
```

不適合例

```
void func(void);
void (*fp)(void) = func; /* NG: &がない */
if (func) { /* NG: 関数呼出しではなく、
アドレスを取得している。引数のない関数呼出しと
勘違いして記述されている場合がある */
```

C言語では、関数名を単独で記述すると関数呼出しではなく、関数アドレス取得となる。すなわち、関数アドレスの取得に&を付ける必要はない。しかしながら、&を付けない場合、関数呼出しと勘違いすることがある（AdaやRubyなど、引数のないサブプログラム呼出しに名前だけを記述する言語を利用している場合など）。関数のアドレスを求める場合に&を付ける規則を守ること、&が付かず、()も続かない関数名の出現をチェックでき、勘違い（ミス）を見つけられる。

〔関連ルール〕

M1.4.2

M1.52

条件判定の式では、0 との比較は明示する。

選択指針

規約化

適合例

```
int x = 5;

if (x != 0) {
  ...
}
```

不適合例

```
int x = 5;

if (x) {
  ...
}
```

条件判定では、式の結果が0の場合は偽、0以外の場合は真と判断される。このため、条件を判定する式では比較演算を省略可能である。しかし、このような記述は、意図しない動作となる可能性がある。よってプログラムの意図を明示するためには、比較を省略しない。また、<stdbool.h>ではbool、true、falseというマクロが定義されているので、真偽を表す型や真偽値を表す定数にはこれらを使用するとよい。

〔関連ルール〕

R2.2.1

M1.4.1

保守性
1.6

領域は1つの利用目的に使用する。

M1.61

目的ごとに変数を用意する。

選択指針

規約化

適合例

```
/* カウンタ変数と入替え用作業変数は別変数 */
for (i = 0; i < MAX; i++) {
  data[i] = i;
}
if (min > max) {
  wk = max;
  max = min;
  min = wk;
}
```

不適合例

```
/* カウンタ変数と入替え用作業変数は同じ変数 */
for (i = 0; i < MAX; i++) {
  data[i] = i;
}
if (min > max) {
  i = max;
  max = min;
  min = i;
}
```

変数の再利用は可読性を損ない、修正時に正しく修正されない危険性が増すので行わない方がよい。

[関連ルール]

M1.2.1



- (1) 共用体は使用してはならない。[MISRA C:2004 18.4]
- (2) 共用体を使用する場合は、書き込んだメンバで参照する。

選択指針	
規約化	選

適合例

```
(2) の適合例
/* type が INT の場合 i_var、
   CHAR の場合 c_var[4] が有効 */
struct stag {
    int type;
    union utag {
        char c_var[4];
        int i_var;
    } u_var;
} s_var;
...
int i;
...
if (s_var.type == INT) {
    s_var.u_var.i_var = 1;
}
...
i = s_var.u_var.i_var;
```

不適合例

```
(2) の不適合例
/* type が INT の場合 i_var、
   CHAR の場合 c_var[4] が有効 */
struct stag {
    int type;
    union utag {
        char c_var[4];
        int i_var;
    } u_var;
} 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;
```

共用体は、1つの領域を異なる大きさの領域で宣言できるが、メンバ間のビットの重なり方が処理系に依存するため、期待する動作にならない可能性がある。使用する場合はルール (2) のような注意が必要である。

[関連ルール]

R2.1.3

名前を再使用しない。

M1.7.1

名前の一意性は、次の規則に従う。

1. 内側のスコープで宣言された識別子は外側のスコープで宣言された識別子を隠してはならない。
[MISRA C:2012 R5.3]
2. typedef 名は一意的な識別子でなければならない。
[MISRA C:2012 R5.6]
3. タグ名は一意的な識別子でなければならない。
[MISRA C:2012 R5.7]
4. 外部結合をもつオブジェクトや関数を定義する識別子は一意的でなければならない。[MISRA C:2012 R5.8]
5. 内部結合をもつオブジェクトや関数を定義する識別子は一意的にするべきである。[MISRA C:2012 R5.9]
6. 構造体及び共用体のメンバ名を除いて、あるネームスペースの識別子を、他のネームスペースの識別子と同じ綴りにしてはいけない。[MISRA C:2004 5.6]

選択指針

○

規約化

適合例

```
int var1;
void func(int arg1) {
    int var2;
    var2 = arg1;
    {
        int var3;
        var3 = var2;
        ...
    }
}
```

不適合例

```
int var1;
void func(int arg1) {
    int var1; /* 関数の外側の変数名と同じ
               名前を使用 */

    var1 = arg1;
    {
        int var1; /* 外側の有効範囲にある変数名と
                     同じ名前を使用 */
        ...
        var1 = 0; /* どのvar1に代入する意図か
                     分からない */
        ...
    }
}
```

■ 名前は、自動変数など有効範囲が限られている場合を除き、できる限りプログラムで一意とすることで、プログラムを読みやすくすることができる。

■ C言語では、名前はファイルやブロックなどによる有効範囲の他に、それが属するカテゴリによって次の4つの名前空間をもっている。

1. ラベル
2. タグ
3. 構造体・共用体のメンバ
4. その他の識別子

※ マクロはネームスペースをもたない

■ ネームスペースが異なれば、言語仕様の同じ名前を付けてもよいが、このルールはそれを制限することで読みやすいプログラムとすることを目的としている。

ルール 2. と 3. の例外として、typedef名はそのtypedefと関連する構造体名、共用体名、タグ名と同じであってもよく、タグ名はそれと関連する typedef名と同じであってもよい。

【 関連ルール 】

M4.3.1



標準ライブラリの関数名、変数名及びマクロ名は再定義・再利用してはならない。また定義を解除してはならない。

選択指針	○
規約化	

適合例

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

不適合例

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

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

標準ライブラリで定義されている関数名や変数名やマクロ名を独自に定義すると、プログラムの可読性を低下させる。

【 関連ルール 】

M1.8.2

M1.73

下線で始まる名前(変数)は定義しない。

選択指針

○

規約化

適合例

—

不適合例

```
int _Max1; /* 予約されている */
int __max2; /* 予約されている */
int _max3; /* 予約されている */

struct S {
    int _mem1; /* 予約されていないが使用しないこととする */
};
```

C 言語規格では、次の名前を予約済みとしている。

- (1) 下線に続き英大文字1字、または下線に続きもう1つの下線で始まる名前

この名前は、いかなる使用に対しても常に予約済みである。

例： _Abc, __abc

- (2) 1つの下線で始まるすべての名前

この名前は、ファイル有効範囲をもつ変数や関数の名前とタグ名に対して予約済みである。

予約済みの名前を再定義した場合、コンパイラの動作が保証されていない。

1つの下線で始まり小文字が続く名前は、ファイル有効範囲以外の部分では予約されていないが、覚えやすいルールとするため、下線で始まる名前すべてを使用しないというルールとしている。

[関連ルール]

M1.82

勘違いしやすい言語仕様を使用しない。

M1.8.1

論理演算子 `&&` または `||` の右側のオペランドには、副作用があってはならない。[MISRA C:2012 R13.5]

選択指針

規約化

適合例

```
volatile int *io_port = ...; /* メモリマップI/O用  
                               アドレス */  
  
int io_result = *io_port;  
/* if文の条件によらずI/O処理実施 */  
if ((x != 0) && (io_result > 0)) {  
    ...  
}
```

不適合例

```
volatile int *io_port = ...; /* メモリマップI/O用  
                               アドレス */  
  
/* if文の条件によってI/O処理を行うか否か異なる */  
if ((x != 0) && (*io_port > 0)) {  
    ...  
}
```

`&&` や `||` 演算子の右式は、左式の条件結果により、実行されない場合がある。インクリメントなどの副作用のある式を右式に記述すると、左式の条件によりインクリメントされる場合とされない場合が生じ、分かりにくくなるため、`&&` や `||` 演算子の右式には副作用のある式を記述しないようにする。

[関連ルール]

R3.6.1, R3.6.2

M1.8.2

Cマクロは、波括弧で囲まれた初期化子、定数、括弧で囲まれた式、型修飾子、記憶域クラス指定子、do-while-zero構造にのみ展開されなければならない。[MISRA C:2004 19.4]

選択指針

規約化

適合例

```
#define START 0x0410  
#define STOP 0x0401
```

不適合例

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

マクロ定義を駆使することにより、C言語以外で書かれたコーディングのように見せかけたり、コーディング量を大幅に減らすことも可能である。しかしながら、このような用途のためにマクロを使用すると可読性が低下する。コーディングミスや変更ミスを防止できる箇所に絞って使用することが重要である。do-while-zeroについては、MISRA C:2004を参照。

[関連ルール]

M1.7.2

M1.8.3

#line は、ツールによる自動生成以外では使用しない。

選択指針

規約化

#line は、コンパイラが出す警告やエラーメッセージのファイル名や行番号を意図的に変更するための機能である。ツールによるコード生成を想定して提供されており、プログラマが直接使うものではない。

M1.8.4

?? で始まる 3 文字以上の文字の並び、及び代替される字句表記は使用しない。

選択指針

規約化

適合例

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

不適合例

```
s = "abc??(x)"; /* トライグラフが可能なコンパイラでは "abc[x]" と解釈される */
```

C 言語規格は、使用している開発環境で利用できない文字があることを想定して、代替の字句表記を規定している。トライグラフと呼ばれる次の 9 つの 3 文字パターン、

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

は、対応する次の 1 文字、

```
#, [, \, ], ^, _ , ~
```

にプリプロセッサの最初で置き換えられる。

ダイグラフと呼ばれる次の文字パターン

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

は、字句解析においてそれぞれ

```
{ } [ ] # ##
```

と同等に扱われる。

C99 では、代替綴りとして、ヘッダ <iso646.h> で次に示すマクロ、

```
and and_eq bitand bitor compl
```

```
not not_eq or or_eq xor xor_eq
```

を定義しており、それぞれ以下の字句に展開される。

```
&& &= & | ^ ! != || |= ^ ^=
```

トライグラフやダイグラフの利用頻度は低いため、オプション指定でサポートしているコンパイラも多い。

M1.85

0で始まる長さ2以上の数字だけの列を定数として使用しない。

選択指針	
規約化	

適合例

```
/* 見た目をよくするための桁揃えはしない */
a = 0;
b = 10;
c = 100;
```

不適合例

```
/* 見た目をよくするために桁を揃えた例 */
a = 000; /* 8進数の0と解釈される */
b = 010; /* 10進数の10でなく8と解釈される */
c = 100; /* 10進数の100と解釈される */
```

0で始まる定数は8進数として解釈される。10進数の見た目の桁を揃えるために、0を前につけること(ゼロパディング)はできない。

[関連ルール]
M1.2.2

保守性1.9 特殊な書き方は意図を明示する。

M1.91

意図的に何もしない文を記述しなければならない場合はコメント、空になるマクロなどを利用し、目立たせる。

選択指針	○
規約化	

適合例

```
for (;;) {
    /* 割込み待ち */
}

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

不適合例

```
for (;;) {
}

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

[関連ルール]
M1.1.1

M1.92

《無限ループの書き方を規定する》

選択指針

○

規約化

規

無限ループの書き方は、例えば以下のような書き方に統一する。

- ・ 無限ループは、for(;;)で統一する。
- ・ 無限ループは、while(1)で統一する。
- ・ マクロ化した無限ループを使用する。

保守性
1.10

マジックナンバーを埋め込まない。

M1.10.1

意味のある定数はマクロとして定義して使用する。

選択指針

○

規約化

適合例

```
#define MAXCNT 8
if (cnt == MAXCNT) {
    ...
}
```

不適合例

```
if (cnt == 8) {
    ...
}
```

マクロ化することにより、定数の意味を明確に示すことができ、定数が複数箇所では使われているプログラムの変更時も1つのマクロを変更すれば済み、変更ミスが防げる。
ただしデータの大きさは、マクロではなく、sizeofを使用する。

〔関連ルール〕

M2.2.4

M1.11.1

参照しかない領域は `const` であることを示す宣言を行う。

選択指針

○

規約化

適合例

```
const volatile int  read_only_mem; /* 参照
  みのメモリ */
const int  constant_data = 10;      /* メモリ
  割付不要な参照のみデータ */
/* argの指す内容を参照するだけ */
void func(const char *arg, int n) {
    int  i;
    for (i = 0; i < n; i++) {
        put(*arg++);
    }
}
```

不適合例

```
int  read_only_mem; /* 参照のみのメモリ */
int  constant_data = 10; /* メモリ割付不要な参
  照のみデータ */
/* argの指す内容を参照するだけ */
void func(char *arg, int n) {
    int  i;
    for (i = 0; i < n; i++) {
        put(*arg++);
    }
}
```

参照するだけで変更しない変数は、`const` 修飾で宣言することで、変更しないことが明確になる。また、コンパイラの最適化処理でオブジェクトサイズが小さくなる可能性もある。このため、参照しかない変数は `const` 修飾にするとよい。また、プログラムからは参照しかないが、他の実行単位からは変更されるメモリは、`const volatile` 修飾で宣言することにより、プログラムで誤って更新することをコンパイラがチェックできる。この他、関数処理内で、引数で示される領域を参照しかない場合にも、`const` を付けることで、関数インタフェースを明示することができる。

[関連ルール]

R1.1.2

M1.11.2

他の実行単位により更新される可能性のある領域は **volatile** であることを示す宣言を行う。

選択指針	○
規約化	

適合例

```
volatile int x = 1;
while (x == 1) {
    /* xはループ内で変更されずに他の実行単位から
       変更される */
}
```

不適合例

```
int x = 1;
while (x == 1) {
    /* xはループ内で変更されずに他の実行単位から
       変更される */
}
```

volatile 修飾された領域は、コンパイラに対し最適化を禁止する。最適化禁止とは、ロジック上は無駄な処理とされる記述に対しても忠実に実行オブジェクトを生成させるということである。

例えば x; という記述があったとする。ロジック的には変数 x を参照するだけで意味のない文のため、volatile 修飾されていなければ、通常コンパイラはこのような記述は無視し、実行オブジェクトは生成しない。volatile 修飾されていた場合は、変数 x の参照（レジスタにロード）だけを行う。この記述の意味するところとしては、メモリをリードするとリセットするようなインターフェースの IO レジスタ（メモリにマッピング）が考えられる。組込みソフトウェアでは、ハードウェアを制御するための IO レジスタがあり、IO レジスタの特性に応じて、適宜 volatile 修飾する必要がある。

M1.11.3

《ROM 化するための変数宣言、定義のルールを規定する。》

選択指針	
規約化	規

適合例

```
const int x = 100; /* ROMに配置 */
```

不適合例

```
int x = 100;
```

const 修飾された変数は ROM 化の対象となる領域に配置することができる。ROM 化を行うプログラムを開発する際には、例えば参照しかならない変数に対して、const 修飾し #pragma などで配置するセクション名を指定する。

〔関連ルール〕

R1.1.2



プリプロセッサが削除する部分でも正しい記述を行う。

選択指針

規約化

適合例

```
#if 0
/* */
#endif

#if 0
...
#else
int var;
#endif

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

不適合例

```
#if 0
/*
#endif

#if 0
...
#else1
int var;
#endif

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

【関連ルール】

M1.1.2

修正誤りのないような書き方にする。

プログラムに不具合が入り込むパターンの1つとして、不具合を修正する際に別の不具合を埋め込んでしまうことがあります。特に、ソースコードを書いたから日時が経っていたり、別の技術者の書いたソースコードを修正する場合、思わぬ勘違いなどが発生することがあります。

こうした修正ミスをできるだけ少なくするための工夫が求められます。

保守性 2.1

構造化されたデータやブロックは、まとまりを明確化する。

保守性 2.2

アクセス範囲や関連するデータは局所化する。

構造化されたデータやブロックは、まとまりを明確化する。

M2.1.1

配列や構造体を0以外で初期化する場合は、構造を示し、それに合わせるために波括弧 "{" を使用しなければならない。また、すべて0以外の場合を除き、データは漏れなく記述する。

選択指針

○

規約化

適合例

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

不適合例

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

配列や構造体の初期化では、最低限、波括弧が一对あればよいが、どのように初期化データが設定されかが分かりにくくなる。構造に合わせてブロック化し、初期化データを漏れなく記述をした方が安全である。

【関連ルール】

R1.2.1

M4.5.3

M2.1.2

if、else if、else、while、do、for、switch
文の本体はブロック化する。

選択指針

規約化

適合例

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

不適合例

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

if文などで制御される文（本体）が複数の文である場合、ブロックで囲む必要がある。制御される文が1つの場合はブロック化する必要はないが、プログラムの変更時に1つの文から複数の文に変更したとき、ブロックで囲み忘れてしまうことがある。変更時のミス未然に防ぐためには、各制御文の本体をブロックで囲むようにする。

アクセス範囲や関連するデータは局所化する。

M2.2.1

1つの関数内でのみ使用する変数は関数内で変数宣言する。

選択指針

●

規約化

適合例

```
void func1(void)
{
    static int x = 0;
    if (x != 0) { /* 前回呼ばれた時の値を参照する */
        x++;
    }
    ...
}
void func2(void)
{
    int y = 0; /* 毎回初期設定する */
    ...
}
```

不適合例

```
int x = 0; /* xはfunc1からしかアクセスされない */
int y = 0; /* yはfunc2からしかアクセスされない */

void func1(void) {
    if (x != 0) { /* 前回呼ばれた時の値を参照する */
        x++;
    }
    ...
}
void func2(void) {
    y = 0; /* 毎回初期設定する */
    ...
}
```

関数内で変数宣言する場合、staticを付けると有効な場合もある。staticを付けた場合、次の特徴がある。

- ・静的領域が確保され、領域はプログラム終了時まで有効（staticを付けないと通常はスタック領域で、関数終了まで有効）。
- ・初期化は、プログラム開始後1度だけで、関数が複数回呼び出される場合、1回前に呼び出されたときの値が保持されている。

このため、その関数内だけでアクセスされる変数のうち、関数終了後も値を保持したいものは、staticを付けて宣言する。また、自動変数に大きな領域を宣言するとスタックオーバーフローの危険がある。そのような場合、関数終了後の値保持が必要なくとも、静的領域を確保する目的で、staticを付けることもある。ただし、この利用方法に対しては、コメントなどで意図を明示することを推奨する（間違えてstaticを付けたと誤解される危険があるため）。

[関連ルール]

M2.2.2



同一ファイル内で定義された複数の関数からアクセスされる変数は、ファイルスコープでstatic変数宣言する。

選択指針	○
規約化	

適合例

```
/* xは他のファイルからアクセスされない*/
static int x;
void func1(void) {
    ...
    x = 0;
    ...
}
void func2(void) {
    ...
    if (x == 0) {
        x++;
    }
    ...
}
```

不適合例

```
/* xは他のファイルからアクセスされない */
int x;
void func1(void) {
    ...
    x = 0;
    ...
}
void func2(void) {
    ...
    if (x==0) {
        x++;
    }
    ...
}
```

グローバルな変数の数が少ないほど、プログラム全体を理解する場合の可読性は向上する。グローバルな変数が増えないように、できるだけstatic記憶クラス指定子を付ける。

【関連ルール】

M2.2.1, M2.2.3



同じファイルで定義した関数からのみ呼ばれる関数は、static関数とする。

選択指針	○
規約化	

適合例

```
/* func1は他のファイルの関数から呼ばれない */
static void func1(void) {
    ...
}
void func2(void) {
    ...
    func1();
    ...
}
```

不適合例

```
/* func1は他のファイルの関数から呼ばれない */
void func1(void) {
    ...
}
void func2(void) {
    ...
    func1();
    ...
}
```

グローバルな関数の数が少ないほど、プログラム全体を理解する場合の可読性は向上する。グローバルな関数が増えないように、できるだけstatic 記憶クラス指定子を付ける。

【 関連ルール 】

M2.2.2

M2.2.4

関連する定数を定義するときは、#define より enum を使用する。

選択指針

規約化

適合例

```
enum ecountry {
    ENGLAND, FRANCE, ...
} country;
enum eweek {
    SUNDAY, MONDAY, ...
} day;
...
if ( country == ENGLAND ) {
if ( day == MONDAY ) {
if ( country == SUNDAY ) { /* ツールで
    チェック可能 */
```

不適合例

```
#define ENGLAND 0
#define FRANCE 1
#define SUNDAY 0
#define MONDAY 1
int country, day;
...
if ( country == ENGLAND ) {
if ( day == MONDAY ) {
if ( country == SUNDAY ) { /* ツールで
    チェック不可 */
```

列挙型は、集合のように関連する定数を定義するときに使用する。関連する定数ごとにenum 型で定義しておくと、誤った使い方に対しツールがチェックすることができるようになる。#define で定義されたマクロ名は、プリプロセス段階でマクロ展開され、コンパイラが処理する名前とならないが、enum 宣言で定義されたenum 定数は、コンパイラが処理する名前となる。コンパイラが処理する名前は、シンボリックデバッグ時に参照できるためデバッグしやすくなる。

【 関連ルール 】

M1.10.1

P1.3.2

プログラムはシンプルに書く。

ソフトウェアの保守しやすさという点に関しては、とにかくソフトウェアがシンプルな書き方になっているに越したことはありません。

C言語は、ファイルに分割する、関数に分割するなどにより、ソフトウェアの構造化が行えます。順次・選択・反復の3つによりプログラム構造を表現する構造化プログラミングもソフトウェアをシンプルに書く技法のひとつです。ソフトウェアの構造化を活用してシンプルなソフトウェア記述を心がけるようにしてください。また、繰り返し処理や代入、演算などについても、書き方によっては保守しにくい形になってしまうため、注意が必要です。

保守性 3.1

構造化プログラミングを行う。

保守性 3.2

1つの文で1つの副作用とする。

保守性 3.3

目的の違う式は、分離して記述する。

保守性 3.4

複雑なポインタ演算は使用しない。

M3.11

繰返し文を終了させるために使用する break 文
または goto 文は、1 つまでとする。

[MISRA C:2012 R15.4]

選択指針

規約化

適合例

```
end = 0;
for (i = 0; ループの継続条件 && !end; i++) {
    繰り出す処理 1;
    if (終了条件 1 || 終了条件 2) {
        end = 1;
    } else {
        繰り出す処理 2;
    }
}
または
for (i = 0; ループの継続条件; i++) {
    繰り出す処理 1;
    if (終了条件 1 || 終了条件 2) {
        break;
    }
    繰り出す処理 2;
}
```

不適合例

```
for (i = 0; ループの継続条件; i++) {
    繰り出す処理 1;
    if (終了条件 1) {
        break;
    }
    if (終了条件 2) {
        break;
    }
    繰り出す処理 2;
}
```

プログラム論理が複雑にならないようにするための工夫である。break 文をなくすためのフラグを用意しなければならない場合は、フラグは用意せずに break を使用した方がよいこともある（適合例の end フラグを使う例は、プログラムを複雑にしてしまう危険があるので要注意）。

M3.12

- (1) goto 文を使用しない。
 (2) goto 文を使用する場合、飛び先は同じブロック、または goto 文を囲むブロック内で、かつ goto 文の後方に宣言されているラベルとする。

選択指針

規約化

選

適合例

(1)、(2) の適合例

```
for (i = 0; ループの継続条件; i++) {
    繰り返す処理;
}
```


 (2) の適合例

```
if (err != 0) {
    goto ERR_RET;
}
...
ERR_RET:
    end_proc();
    return err;
}
```

不適合例

(1)、(2) の不適合例

```
i = 0;
LOOP:
    繰り返す処理;
    i++;
    if (ループの継続条件) {
        goto LOOP;
    }
```

プログラム論理が複雑にならないようにするための工夫である。goto 文をなくすことが目的ではなく、プログラムが複雑になる（上から下へ読めなくなる）ことを避けるための手段として、不要な goto をなくす、ということが重要である。goto 文を用いることにより、可読性が向上することもある。いかに論理をシンプルに表現できるかを考えて、プログラムする。

なお、プログラムをシンプルにするために、goto 文を使用する場合として、例えば、エラー処理への分岐や多重ループを抜ける場合などがある。

M3.14

- (1) switch文のcase節、default節は、必ずbreak文で終了させる。
- (2) switch文のcase節、default節をbreak文で終了させない場合は、《プロジェクトでコメントを規定し》そのコメントを挿入する。

選択指針	○
規約化	選規

適合例

```
(1)、(2) の適合例
switch (week) {
case A:
    code = MON;
    break;
case B:
    code = TUE;
    break;
case C:
    code = WED;
    break;
default:
    code = ELSE;
    break;
}
```

```
(2) の適合例
dd = 0;
switch (status) {
case A:
    dd++;
    /* FALL THROUGH */
case B:
```

不適合例

```
(1)、(2) の不適合例
/* weekがどんな値でも、codeはELSEになってしまう
   ==> バグ */
switch (week) {
case A:
    code = MON;
case B:
    code = TUE;
case C:
    code = WED;
default:
    code = ELSE;
}
```

```
/* case Bの処理を継続してよい場合だが、コメントがないので(1)だけでなく(2)にも不適合 */
dd = 0;
switch (status) {
case A:
    dd++;
case B:
```

C言語のswitch文におけるbreak忘れは、コーディングミスしやすい代表例の1つである。不要に、breakなしのcase文を使用することは避けるべきである。break文なしに次のcaseに処理を継続させる場合は、break文がなくても問題のないことを明示するコメントを、必ず入れるようにする。どのようなコメントを入れるかは、コーディング規約で定める。例えば、`/* FALL THROUGH */`などがよく利用される。

[関連ルール]

R3.5.2

M3.15

- (1) 関数は、1つのreturn文で終了させる。
- (2) 処理の途中で復帰するreturn文は、異常復帰の場合のみとする。

選択指針	
規約化	選

プログラム論理が複雑にならないようにするための工夫である。プログラムの入口や出口が沢山あると、プログラムを複雑にするだけでなく、デバッグする時も、ブレークポイントを入れるのが大変になる。C言語の場合、関数の入口は1つであるが、出口はreturn文を書いたところとなる。

保守性 3.2

1つの文で1つの副作用とする。

M3.2.1

- (1) コンマ式は使用しない。
- (2) コンマ式はfor文の初期化式や更新式以外では使用しない。

選択指針	
規約化	選

適合例

(1)、(2)の適合例

```
a = 1;
b = 1;
```

```
j = 10;
for (i = 0; i < 10; i++) {
    ...
    j--;
}
```

(2)の適合例

```
for (i = 0, j = 10; i < 10; i++, j--) {
    ...
}
```

不適合例

(1)、(2)の不適合例

```
a = 1, b = 1;
```

(1)の不適合例

```
for (i = 0, j = 10; i < 10; i++, j--) {
    ...
}
```

コンマ式を利用すると複雑になる。ただし、for文の初期化式や更新式は、ループの前後に実施すべき処理をまとめて記述する場所であり、コンマ式を利用してまとめて記述する方がわかりやすいこともある。

【関連ルール】

M3.3.1

M3.2.2

1つの文に、代入を複数記述しない。ただし、同じ値を複数の変数に代入する場合を除く。

選択指針	○
規約化	

適合例

```
x = y = 0;
```

不適合例

```
y = (x += 1) + 2;  
y = (a++) + (b++);
```

代入には、単純代入(=)の他に、複合代入(+=、-= など)がある。1つの文に複数の代入を記述できるが、可読性を低下させるため、1つの文では1つの代入にとどめるべきである。

ただし、次の「よく使用される慣習的な記述」については、可読性を損なわない場合も多い。例外として許すルールとしてもよい。

```
c = *p++;  
*p++ = *q++;
```

保守性
3.3

目的の違う式は、分離して記述する。

M3.3.1

for文の3つの式には、ループ制御に関わるもののみを記述しなければならない。[MISRA C:2004 13.5]

選択指針	
規約化	

適合例

```
for (i = 0; i < MAX; i++) {  
    ...  
    j++;  
}
```

不適合例

```
for (i = 0; i < MAX; i++, j++) {  
    ...  
}
```

MISRA C:2012では、MISRA C:2004 13.5、13.6はまとめられ、「R14.2 for ループは適格 (well-formed) でなければならない。」となった。このルールは、例えば最初の式の場合、空か、forループのカウンタに値を設定するか、ループカウンタを定義(初期化含む)する(C99の場合)のいずれかでなければならないというように、詳細が規定されている。

[関連ルール]

M3.2.1, M3.3.2



forループの中で繰返しカウンタとして用いる数値変数は、ループの本体内で変更してはならない。
[MISRA C:2004 13.6]

選択指針	
規約化	

適合例

```
for (i = 0; i < MAX; i++) {  
    ...  
}
```

不適合例

```
for (i = 0; i < MAX; ) {  
    ...  
    i++;  
}
```

M3.3.1 参照

[関連ルール]

M3.3.1



(1) 真偽を求める式の中で代入演算子を使用しない。
(2) 真偽を求める式の中で代入演算子を使用しない。ただし慣習的に使う表現は除く。

選択指針	
規約化	選

適合例

```
(1)、(2) の適合例  
p = top_p;  
if (p != NULL) {  
    ...  
}  
  
(1) の適合例  
c = *p++;  
while (c != '\0') {  
    ...  
    c = *p++;  
}
```

不適合例

```
(1)、(2) の不適合例  
if (p = top_p) {  
    ...  
}  
  
(1) の不適合例  
while (c = *p++) {  
    ...  
}  
/* 慣習的に使う表現なので (2) では OK  
   (開発者のスキルに依存するので要注意) */
```

真偽を求める式は、以下の式である。

if(式)、for(;式;)、while(式)、(式)? : 、
式 && 式、式 || 式

複雑なポインタ演算は使用しない。

M3.41

3段階以上のポインタ指定は使用しない。

選択指針

規約化

適合例

```
int **p;  
typedef char **strptr_t;  
strptr_t q;
```

不適合例

```
int ***p;  
typedef char **strptr_t;  
strptr_t *q;
```

- 3段階以上のポインタの値の変化を理解することは難しいため、保守性を損なう。

統一した書き方にする。

最近のプログラム開発では、複数人による分業開発が定着しています。このような場合、開発者それぞれが異なったソースコードの書き方をしていると、それぞれの内容確認を目的としたレビューなどがしづらいといった問題が発生します。また、変数のネーミングやファイル内の情報の記載内容や記載順序などがバラバラだと、思わぬ誤解や誤りのもとになりかねません。このため、1つのプロジェクトや組織内では、極力ソースコードの書き方を統一しておいた方がよいとされています。

保守性 4.1

コーディングスタイルを統一する。

保守性 4.2

コメントの書き方を統一する。

保守性 4.3

名前の付け方を統一する。

保守性 4.4

ファイル内の記述内容と記述順序を統一する。

保守性 4.5

宣言の書き方を統一する。

保守性 4.6

ナルポインタの書き方を統一する。

保守性 4.7

前処理指令の書き方を統一する。

M4.11

《波括弧 ({ }) や字下げ、空白の入れ方などのスタイルに関する規約を規定する。》

選択指針

○

規約化

規

コードの見やすさのために、コーディングスタイルをプロジェクトで統一することは重要である。スタイルの規約をプロジェクトで新規に決定する場合、世の中ですでに存在するコーディングスタイルから選択することを推奨する。既存のコーディングスタイルには、いくつかの流派があるが、多くのプログラマがそのいずれかに沿ってプログラムを作成している。それらのスタイルを選択することで、エディタやプログラムの整形コマンドなどで簡単に指定できるといった恩恵も受けられる。一方、既存のプロジェクトでコーディングスタイルが明確に示されていない場合、現状のソースコードに一番近い形で規約を作成することを推奨する。

スタイル規約の決定において、最も重要なことは「決定して統一する」ことであり、「どのようなスタイルに決定するか」ではないことに注意すること。

以下、決めるべき項目について説明する。

(1) 波括弧 ({ }) の位置

波括弧の位置は、ブロックの始まりと終わりを見やすくするために統一する (「代表的なスタイル」参照)。

(2) 字下げ (インデントーション)

字下げは、宣言や処理のまとまりを見やすくするために行う。字下げの統一で規定することは次の通り。

- ・字下げに、空白を使用するか、タブを使用するか。
- ・空白の場合、空白を何文字とするか、タブの場合、1タブを何文字とするか。

(3) 空白の入れ方

空白はコードを見やすくし、また、コーディングミスを発見しやすくするために挿入する。例えば、次のようなルールを規定する。

- ・2項演算子、および3項演算子の前後に空白を入れる。ただし、次の演算子を除く。
[], ->, . (ピリオド), , (コンマ演算子)
- ・単項演算子と演算項の間には、空白を入れない。

これらのルールは、複合代入演算子のコーディングミスを発見しやすくする。

[例]

```
x=-1;      /* x-=1 と書くのを誤って書いてしまった ⇒ 見つけにくい */
x -= 1;    /* x-=1 と書くのを誤って書いてしまった ⇒ 見つけやすい */
```

上記の他に、以下のようなルールを定めることもある。

- ・コンマの後に空白を入れる（ただし、マクロ定義時の引数のコンマを除く）。
- ・ifやforなどの制御式を囲む左括弧の前に空白を入れる。関数呼出しの左括弧の前には空白を入れない。このルールは、関数呼出しを探しやすくする。

(4) 継続行における改行の位置

式が長くなり、見やすい1行の長さを超える場合、適当な位置で改行する。改行にあたっては、次の2つの方式のいずれかを選択することを推奨する。重要なことは、継続行は字下げして記述することである。

[方式1] 演算子を行の最後に書く

```
例：
x = var1 + var2 + var3 + var4 +
    var5 + var6 + var7 + var8 + var9;
if (var1 == var2 &&
    var3 == var4)
```

[方式2] 演算子を継続行の先頭に書く

```
例：
x = var1 + var2 + var3 + var4
    + var5 + var6 + var7 + var8 + var9;
if (var1 == var2
    && var3 == var4)
```

●代表的なスタイル

(1) K&Rスタイル

『プログラミング言語C』（略称 K&R 本）で用いられたコーディングスタイルである。この本の2人の著者のイニシャルから、本の略称同様このように呼ばれている。K&Rスタイルにおける波括弧の位置、字下げは次の通り。

- ・波括弧の位置 関数定義の波括弧は、改行して行の先頭に記述する。その他（構造体、if、for、whileなどの制御文など）は、改行なしでその行に記述する（例参照）。
- ・字下げ 1タブ。『プログラミング言語C』初版では5だったが、第2版(ANSI対応版)では4。

(2) BSD スタイル

多くのBSDのユーティリティを記述したEric Allman氏の記述スタイルで、Allmanスタイルとも呼ばれている。BSDスタイルにおける、波括弧の位置、字下げは次の通り。

- ・波括弧の位置 関数定義、if、for、whileなどすべて改行し、波括弧は、前の行と揃えたコラムに置く（例参照）。
- ・字下げ 8とする。4も多い。

(3) GNU スタイル

GNUパッケージを記述するためのコーディングスタイルである。Richard Stallman氏とGNUプロジェクトのボランティアの人々によって書かれた「GNUコーディング規約」で定められている。GNUスタイルにおける、波括弧の位置、字下げは次の通り。

- ・波括弧の位置 関数定義、if、for、whileなどすべて改行し記述する。関数定義の波括弧は、コラム0に置き、それ以外は、2文字分の空白をいれたインデントとする（例参照）。
- ・字下げ 2。波括弧、本体ともに、2インデントする。

(1) K&Rスタイルの例：

```
void func(int arg1)
{ /* 関数の { は改行して記述する */
  /* インデントは1タブ */
  if (arg1) {
    ...
  }
  ...
}
```

(2) BSDスタイルの例：

```
void
func(int arg1)
{ /* 関数の { は改行して記述する */
  if (arg1)
  {
    ...
  }
  ...
}
```

(3) GNUスタイルの例：

```
void
func(int arg1)
{ /* 関数の { は改行してコラム0に記述する */
  if (arg1)
  {
    ...
  }
  ...
}
```

M4.2.1

《ファイルヘッダコメント、関数ヘッダコメント、行末コメント、ブロックコメント、コピーライトなどの書き方に関する規約を規定する。》

選択指針

○

規約化

規

コメントを上手に記述することで、プログラムの可読性が向上する。さらに見やすくするためには、統一した書き方が必要となる。

また、ソースコードから保守・調査用ドキュメントを生成するドキュメント生成ツールがある。このようなツールを活用する場合、その仕様に沿った書き方が必要になる。ドキュメント生成ツールは、一般的には、変数や関数の説明を一定のコメント規約の下で記述すると、ソースコードからドキュメントに反映される。ツールの仕様を調査して、コメント規約を定めるべきである。

以下に、既存のコーディング規約や書籍などから、コメントの書き方に関するものを紹介する。

●代表的なコメントの書き方

(1) Indian Hill コーディング規約

Indian Hill コーディング規約では、次のようなコメントルールが記述されている。

・ブロックコメント

データ構造やアルゴリズムを記述する場合に利用する。形式は、1桁目に / を書き、すべて2桁目に * を書き、最後は、2桁目と3桁目に */ を書く (grep ^/* でブロックコメントが抽出可能)。

例：

```
/* コメントを書く  
 * コメントを書く  
*/
```

・コメントの位置

- 関数中のブロックコメント

次の行の字下げ位置に揃えるなど、ふさわしい位置に書く。

- 行末のコメント

タブで遠く離して書く。複数のそのようなコメントがある場合は、同じ字下げ位置に揃える。

(2) GNU コーディング規約

GNU コーディング規約では、次のようなコメントルールが記述されている。

・記述言語

- 英語

- ・記述場所と内容

- プログラムの先頭

- すべてのプログラムは、何をするプログラムかを簡潔に説明するコメントで始める。

- 関数

- 関数ごとに次のコメントを書く。

- 何をする関数か、引数の説明（値、意味、用途）、戻り値

- #endif

- 入れ子になっていない短い条件を除いて、#endifはすべて行末にコメントを入れ、条件を明確にする。

- ツール用の記法

- コメントのセンテンスの終わりには、空白を2文字置く。

(3) "プログラミング作法"

"プログラミング作法"では、次のようなコメントルールが記述されている。

- ・記述場所

- 関数とグローバルデータに対して記述する。

- ・その他の作法

- 当たり前のことはいちいち書かない。

- コードと矛盾させない。

- あくまでも明快に、混乱を招く記述をしない。

(4) その他

- ・ /* */ コメントと // コメントの使用方針を決める。

例1：文末コメントには「//」を、ブロックコメントには「/* */」を使用する。

例2：「/* */」は閉じ忘れの恐れがあるので、「//」を使用する。

例3：コメント内で「/*」や「//」は使用しない。ただし、//コメント中の「//」は例外とする。

- ・著作権表示をコメント内に記述する。

- ・break なし case 文に対するコメントを決める。

```
例：
switch (status) {
case CASE1:
    処理;
    /* FALL THROUGH */
case CASE2:
    ...
}
```

- ・処理なしに対するコメントを決める。

```
例：
if (条件1) {
    処理;
} else if (条件2) {
    処理;
} else {
    /* DO NOTHING */
}
```

- ・ // コメント内で行連結を行ってはならない。(MISRA C:2012 R3.2)

名前の付け方を統一する。

M4.3.1

《外部変数、内部変数などの命名に関する規約を規定する。》

選択指針

○

規約化

規

☆名前の付け方について を参照のこと。

[関連ルール]

M1.7.1, M1.7.2, M1.7.3, M4.3.2, P1.1.2, P1.2.1

M4.3.2

《ファイル名の命名に関する規約を規定する。》

選択指針

○

規約化

規

☆名前の付け方について を参照のこと。

[関連ルール]

M4.3.1, P1.1.2, P1.2.1

☆名前の付け方について

プログラムの読みやすさは、名前の付け方に大きく左右される。名前の付け方にも様々な方式があるが、重要なことは統一性であり、分かりやすさである。

名前の付け方では、次のような項目を規定する。

- ・ 名前全体に対する指針
- ・ ファイル名（フォルダ名やディレクトリ名を含む）の付け方
- ・ グローバルな名前とローカルな名前の付け方
- ・ マクロ名の付け方、など

以下に、既存のコーディング規約や書籍などから紹介されている名前の付け方に関する指針やルールの幾つかを紹介する。プロジェクトで新規に命名規約を作成する場合の参考にとよい。既存のプロジェクトで命名規約が明確に示されていない場合、現状のソースコードに一番近い規約を作成することを推奨する。

●代表的な命名規約

(1) Indian Hill コーディング規約

- ・下線が前後についた名前はシステム用に確保してあるので使用しない。
- ・`#define` される定数名はすべて大文字にする。
- ・`enum` で定義する定数は、先頭もしくはすべての文字を大文字にする。
- ・大文字と小文字の差しかない2つの名前は使用しない (`foo` と `Foo` など)。
- ・グローバルなものには、どのモジュールに属するのか分かるように、共通の接頭辞を付ける。
- ・ファイル名のはじめの文字は英字、その後の文字は英数字で8文字以下(拡張子を除く)とする。
- ・ライブラリのヘッダファイルと同じ名前のファイル名は避ける。

全体		<ul style="list-style-type: none">・下線が前後に付いた名前は使用しない・大文字と小文字の差しかない2つの名前は利用しない 例: <code>foo</code> と <code>Foo</code>
変数名、 関数名	グローバル	モジュール名の接頭辞を付ける
	ローカル	特になし
その他		<ul style="list-style-type: none">・マクロ名は、すべてを大文字とする 例: <code>#define MACRO</code>・<code>enum</code> メンバは、先頭もしくはすべての文字を大文字とする。

(2) GNU コーディング規約

- ・グローバル変数や関数の名前は、短すぎる名前を付けない。英語で意味をもつような名前にする。
- ・名前の中で語を分けるのには下線を使う。
- ・大文字は、マクロと `enum` 定数と一定の規約に従った名前の接頭辞にのみ使用する。通常は、英小文字のみの名前を使用する。

全体		<ul style="list-style-type: none">・名前の中で語を分けるのに下線を使う 例: <code>get_name</code>・大文字は、マクロと <code>enum</code> 定数と一定の規約に従った名前の接頭辞にのみ使用する。通常は、英小文字のみの名前を使用する
変数名、 関数名	グローバル	短すぎる名前を付けない。英語で意味をもつ名前とする
	ローカル	特になし
その他		<ul style="list-style-type: none">・マクロ名は、すべてを大文字とする 例: <code>#define MACRO</code>・<code>enum</code> メンバは、すべての文字を大文字とする

(3) "プログラミング作法"

- ・グローバルには分かりやすい名前を、ローカルには短い名前を付ける。
- ・関連性のあるものには、関連性のある名前を付けて、違いを際立たせるようにする。
- ・関数名は能動的な動詞を基本にし、特に問題がなければその後に関数を付ける。

全体		関連性のあるものには、関連性のある名前を付ける
変数名、 関数名	グローバル	分かりやすい名前を付ける
	ローカル	短い名前を付ける
	その他	関数名は、能動的な動詞を基本にし、特に問題がなければその後に関数を付ける

(4) その他

- ・次の違いしかない2つの名前は使用しない
 - 下線の有無
 - 文字O、D、あるいは数字0
 - 文字I、(I 小文字エル)、あるいは数字1
 - 文字S、あるいは数字5
 - 文字Z、あるいは数字2
 - 文字n、あるいはh
- ・名前の区切り方 複数の単語で構成される名前の単語の区切りは、下線で区切るか単語の1文字目を大文字にして区切るか、どちらかに統一する。
- ・ハンガリアン記法 変数の型を明示的にするためのハンガリアン記法などもある。
- ・ファイル名の付け方 接頭辞として、例えばサブシステムを示す名前を付ける。

保守性 4.4

ファイル内の記述内容と記述順序を統一する。



《ヘッダファイルに記述する内容（宣言、定義など）とその記述順序を規定する。》

選択指針

○

規約化

規

複数の箇所に記述すると変更ミスの危険があるため、共通に使用するものはヘッダファイルに記述する。ヘッダファイルには、複数のソースファイルで共通に使用するマクロの定義、構造体・共用体・列挙型のタグ宣言、typedef宣言、外部変数宣言、関数プロトタイプ宣言を記述する。例えば、以下の順序で記述する。

- (1) ファイルヘッダコメント
- (2) システムヘッダの取り込み

- (3) ユーザ作成ヘッダの取り込み
- (4) #define マクロ
- (5) #define 関数マクロ
- (6) typedef 定義 (int や char といった基本型に対する型定義)
- (7) enum タグ定義 (typedef を同時に行う)
- (8) struct/union タグ定義 (typedef を同時に行う)
- (9) extern 変数宣言
- (10) 関数プロトタイプ宣言
- (11) インライン関数

M4.42

《ソースファイルに記述する内容 (宣言、定義など) とその記述順序を規定する。》

選択指針

○

規約化

規

ソースファイルには、変数及び関数の定義と、個々のソースファイルでのみ使用するマクロ、タグ、型 (typedef 型) の定義や宣言を記述する。

例えば、以下の順序で記述する。

- (1) ファイルヘッダコメント
- (2) システムヘッダの取り込み
- (3) ユーザ作成ヘッダの取り込み
- (4) 自ファイル内でのみ使用する #define マクロ
- (5) 自ファイル内でのみ使用する #define 関数マクロ
- (6) 自ファイル内でのみ使用する typedef 定義
- (7) 自ファイル内でのみ使用する enum タグ定義
- (8) 自ファイル内でのみ使用する struct/union タグ定義
- (9) ファイル内で共有する static 変数宣言
- (10) static 関数宣言
- (11) 変数定義
- (12) 関数定義

※ (2)、(3) は不要なものを取り込まないように気を付ける。

※ (4) ～ (8) はできるだけ記述しない。

M4.43

外部変数や関数（ファイル内でのみ使用する関数を除く）を使用したり定義する場合、宣言を記述したヘッダファイルをインクルードする。

選択指針	○
規約化	

適合例

```
--- my_inc.h ---
extern int x;
int func(int);

-----
#include "my_inc.h"
int x;
int func(int in)
{
    ...
}
```

不適合例

```
/* 変数 x や関数 func の宣言がない */
int x;
int func(int in)
{
    ...
}
```

C言語では、変数は使用前に宣言か定義が必要である。一方、関数は宣言も定義もなくとも使用できる。しかしながら、宣言と定義の整合性を保つために、宣言をヘッダファイルに記述し、そのヘッダファイルをインクルードすることを推奨する。

M4.44

外部変数は、複数箇所で定義しない。

選択指針	●
規約化	

適合例

```
int x;    /* 1つの外部変数定義は、1つにする */
```

不適合例

```
int x;
int x;    /* 外部変数定義は複数箇所でも
コンパイルエラーにならない */
```

外部変数は、初期化なしの定義を複数記述できる。ただし、複数ファイルで初期化を行った場合の動作は保証されない。

M4.45

ヘッダファイルには、変数定義や関数定義を記述しない。

選択指針

○

規約化

適合例

```
--- file1.h ---
extern int x; /* 変数宣言 */
int func(void); /* 関数宣言 */

--- file1.c ---
#include "file1.h"
int x; /* 変数定義 */
int func(void) /* 関数定義 */
{
    ...
}
```

不適合例

```
--- file1.h ---
int x; /* 外部変数定義 */
static int func(void) /* 関数定義 */
{
    ...
}
```

ヘッダファイルは、複数のソースファイルに取り込まれる可能性がある。このため、ヘッダファイルに変数定義や関数定義を記述すると、コンパイル後に生成されるオブジェクトコードのサイズが不必要に大きくなる危険がある。ヘッダファイルは、基本的に宣言や型の定義などに限定して記述する。

M4.46

ヘッダファイルは重複取り込みに耐えうる作りとする。《そのための記述方法を規定する。》

選択指針

○

規約化

規

適合例

```
--- myheader.h ---
#ifndef MYHEADER_H
#define MYHEADER_H
    ヘッダファイルの内容
#endif /* MYHEADER_H */
```

不適合例

```
--- myheader.h ---
void func(void);
/* end of file */
```

ヘッダファイルは、重複して取り込む必要がないようにできる限り整理しておくべきである。しかしながら、重複して取り込まれる場合もある。そのためのために、重複取り込みに耐えうる作りとすることも必要である。

例えば、次のルールとする。

ルールの例：

ヘッダファイルの先頭で、ヘッダを取り込み済みか否かを判定する `#ifndef` マクロを記述し、2 回目の取り込みでは、以降の記述がコンパイル対象にならないようにする。この時のマクロ名は、ヘッダファイルの名前をすべて大文字にした名前、ピリオドを `_`（下線）に変更した名前を付ける。

宣言の書き方を統一する。

M4.5.1

- (1) 関数プロトタイプ宣言では、すべての引数に名前を付けない（型だけとする）。
- (2) 関数プロトタイプ宣言では、すべての引数に名前を付ける。さらに、引数の型と名前、及び戻り型は、関数定義と文字通りに同じにする。

選択指針

○

規約化

選

適合例

(1) の適合例

```
int func1(int, int);
```

```
int func1(int x, int y)
```

```
{  
    /* 関数の処理 */  
}
```

(2) の適合例

```
int func1(int x, int y);  
int func2(float x, int y);
```

```
int func1(int x, int y)  
{  
    /* 関数の処理 */  
}
```

```
int func2(float x, int y)  
{  
    /* 関数の処理 */  
}
```

不適合例

(1)、(2) の不適合例

```
int func1(int x, int y);  
int func2(float x, int y);
```

```
int func1(int y, int x) /* 引数の名前がプロト  
                        タイプ宣言と異なる */
```

```
{  
    /* 関数の処理 */  
}
```

```
typedef int INT;  
int func2(float x, INT y) /* yの型がプロトタイプ  
                        宣言と文字通りに同じでない */
```

```
{  
    /* 関数の処理 */  
}
```

関数プロトタイプ宣言では引数の名前を省略できるが、適切な引数名の記述は関数インタフェース情報として価値がある。引数名を記述する場合、定義と同じ名前を利用し、無用な混乱を避けるべきである。

型についても、関数定義と文字通りに同じ方が理解が容易であり、同じにすることを推奨する。

また、引数が特定の大きさの配列の場合は、その要素数を指定することが望ましい。

例: `void func(int a[4]) {…}`

[関連ルール]

M1.4.1

M4.5.2

構造体タグの宣言と変数の宣言は別々に行う。

選択指針

規約化

適合例

```
struct TAG {
    int mem1;
    int mem2;
};
struct TAG x;
```

不適合例

```
struct TAG {
    int mem1;
    int mem2;
} x;
```

M4.5.3

- (1) 構造体・共用体・配列の初期値式のリスト、及び列挙子リストの最後の"}"の前に","を記述しない。
- (2) 構造体・共用体・配列の初期値式のリスト、及び列挙子リストの最後の"}"の前に","を記述しない。ただし、配列の初期化の初期値リストの最後の"}"の前に","を書くことは許す。

選択指針

規約化

選

適合例

```
(1) の適合例
struct tag data[] = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 } /* 最後の要素にコンマはない */
};
(2) の適合例
struct tag data[] = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }, /* 最後の要素にコンマがある */
};
```

不適合例

```
(1)、(2) の不適合例
struct tag x = { 1, 2, };
/* メンバが2つだけなのか、3つ以上あるのか不明確 */
```

複数のデータの初期化では、初期化の最後を明示するために、最後の初期値にコンマをつけない流派と、初期値の追加・削除のしやすさを考慮し、最後にコンマをつける流派がある。どちらを重視するかを検討し、ルールを決定する。

なお、C90の規格では、列挙子リストの最後を示す"}"の直前の","は許されなかったが、C99の規格では、許されるようになった。

【関連ルール】

M2.1.1

ナルポインタの書き方を統一する。

M4.6.1

- (1) ナルポインタには0を使用する。NULLはいかなる場合にも使用しない。
- (2) ナルポインタにはNULLを使用する。NULLはナルポインタ以外に使用しない。

選択指針

○

規約化

選

適合例

(1) の適合例

```
char *p;  
int dat[10];
```

```
p = 0;  
dat[0] = 0;
```

(2) の適合例

```
char *p;  
int dat[10];
```

```
p = NULL;  
dat[0] = 0;
```

不適合例

(1) の不適合例

```
char *p;  
int dat[10];
```

```
p = NULL;  
dat[0] = NULL;
```

(2) の不適合例

```
char *p;  
int dat[10];
```

```
p = 0;  
dat[0] = NULL;
```

- NULLはナルポインタとして従来使用されてきた表現だが、実行環境によりナルポインタの表現は異なる。このため、0を使う方が安全だと考える人もいる。

前処理指令の書き方を統一する。



演算子を含むマクロは、マクロ本体とマクロ引数を括弧で囲む。

選択指針	○
規約化	

適合例

```
#define M_SAMPLE(a, b) ((a)+(b))
```

不適合例

```
#define M_SAMPLE(a, b) a+b
```

- マクロ本体、マクロ引数を括弧で囲っていない場合、マクロ展開後にマクロに隣接する演算子とマクロ内の演算子の優先順位によって、意図する演算順序にならずバグになる可能性がある。



#ifdef、#ifndef、#ifに対応する#else、#elif、#endifは、同一ファイル内に記述し、《プロジェクトで規定したコメントを入れ対応関係を明確にする。》

選択指針	○
規約化	規

適合例

```
#ifdef AAA
/* AAAが定義されたときの処理 */
...
#else /* not AAA */
/* AAAが定義されないときの処理 */
...
#endif /* end AAA */
```

不適合例

```
#ifdef AAA
/* AAAが定義されたときの処理 */
...
#else
/* AAAが定義されないときの処理 */
...
#endif
```

- #ifdefなどマクロによる処理の切り分けにおいて、#elseや#endifが離れた場所に記述されたりネストすると対応が分かりにくくなる。#ifdefなどに対応する#else、#endifなどにコメントを付け、対応を分かりやすくする。

【関連ルール】

M1.1.1, M1.1.2



#if や #elif で、マクロ名が定義済みかを調べる場合は、defined(マクロ名) または defined マクロ名により定義済みかを調べる。

選択指針	●
規約化	

適合例

```
#if defined(AAA)
...
#endif
または
#if defined AAA
...
#endif
```

不適合例

```
#if AAA
...
#endif
または
#define DD(x) defined(x)
#if DD(AAA)
...
#endif
```

#ifマクロ名としても、マクロが定義されているかどうかの判定にはならない。例えば #if AAA の場合、マクロ AAA が定義されていない場合だけでなく、マクロ AAA の値が0 の場合も判定は偽となる。マクロが定義されているかどうかをチェックするには、defined を利用すべきである。

defined の記述において、defined(マクロ名) または definedマクロ名以外の書き方をした場合、C 言語規格ではどのように処理されるか定義していない (未定義)。コンパイラによってエラーにしたり、独自の解釈をしている場合があるため、使用しないようにする。

[関連ルール]

M4.7.7



マクロは、ブロック内で #define、または #undef してはならない。[MISRA C:2004 19.5]

選択指針	
規約化	

適合例

```
#define AAA 0
#define BBB 1
#define CCC 2
struct stag {
    int mem1;
    char *mem2;
};
```

不適合例

```
/* 設定される値に制限があるメンバが存在する */
struct stag {
    int mem1; /* 以下の値が設定可能 */
#define AAA 0
#define BBB 1
#define CCC 2
    char *mem2;
};
```

マクロ定義（#define）は、一般的にファイルの先頭にまとめて記述する。ブロック内に記述するなど、分散させると可読性が低下する。また、ブロック内で定義を解除（#undef）することも可読性を低下させる。マクロ定義は変数と違い、有効範囲はファイルの終わりまでになることに注意する。不適合例のプログラムは次のように変更することも考えられる。

```
enum etag { AAA, BBB, CCC };
struct stag {
    enum etag    mem1;
    char         *mem2;
};
```

【 関連ルール 】

M4.7.6

M4.7.6

#undefは使用してはならない。

[MISRA C:2012 R.20.5]

選択指針

規約化

#defineしたマクロ名は、#undefすることにより、定義されていない状態にすることが可能であるが、マクロ名が参照されている箇所により、解釈が異なる危険があり、可読性を低下させる。

【 関連ルール 】

M4.7.5

M4.77

#if または #elif 前処理指令の制御式は、0 または 1 に評価されなければならない。

[MISRA C:2012 R20.8]

選択指針

規約化

適合例

```
#define TRUE 1
#define FALSE 0
#if TRUE
...

#if defined(AAA)
...

#if VERSION == 2
...

#if 0 /* ~のため、無効化 */
...
```

不適合例

```
#define ABC 2
#if ABC
...
```

#if 制御式、#elif 制御式は、制御式が真か偽を評価する。制御式には真か偽かが分かり易い記述をし、プログラムを読みやすくする。

【関連ルール】

M4.7.3

試験しやすい書き方にする。

組込みソフトウェア開発で欠くことのできない作業の1つが動作確認(テスト)です。しかし、近年の複雑な組込みソフトウェアでは、テスト時に検出された不具合や誤動作が再現できないなどの問題が発生する場合があります。このため、ソースコードを記述する際に、問題原因分析のしやすさなどまで考慮しておくことが望まれます。また、特に、動的メモリの利用などはメモリリークなどの危険があるため、特別な注意を払う必要があります。

保守性 5.1

問題発生時の原因を調査しやすい書き方にする。

保守性 5.2

動的なメモリ割り当ての使用に気を付ける。

M5.11

《デバッグオプション設定時のコーディング方法と、リリースモジュールにログを残すためのコーディング方法を規定する。》

選択指針

○

規約化

規

プログラムは、所定の機能を実装するだけではなく、デバッグや問題発生時の調査のしやすさを考慮に入れて、コーディングする必要がある。問題を調査しやすくするための記述には、リリース用モジュールには反映されないデバッグ用記述と、リリース用モジュールにも反映するリリース後のログ出力の記述がある。以下、それぞれに関する規約の決め方について説明する。

●デバッグ用記述

プログラム開発中に利用する print 文など、デバッグ用の記述は、リリース用モジュールには反映しないように記述する。ここでは、デバッグ用記述の方法として、マクロ定義を利用した切り分けと assert マクロの利用について説明する。

(a) デバッグ処理記述の切り分けに、マクロ定義を利用する方法

デバッグ用の記述を提供モジュールに反映しないようにするためには、コンパイル対象の有無を切り分けるマクロ定義を利用する。マクロ名には、文字列 "DEBUG" または、"MODULEA_DEBUG" など "DEBUG" を含む名前がよく用いられる。

ルール定義の例：

デバッグ用のコードは、`#ifdef DEBUG` で切り分ける (DEBUG マクロは、コンパイル時に指定する)。

【コード例】

```
#ifdef DEBUG
fprintf(stderr, "var1 = %d\n", var1);
#endif
```

さらに、次のようなマクロ定義を利用する方法もある。

ルール定義の例：

デバッグ用のコードは、`#ifdef DEBUG` で切り分ける (DEBUG マクロは、コンパイル時に指定する)。

さらにデバッグ情報出力では、次のマクロを利用する。

```
DEBUG_PRINT(str); /* str を標準出力に出力する */
```

このマクロは、プロジェクトの共通ヘッダ `debug_macros.h` に定義しているため、利用時には、そのヘッ

ダをインクルードする。

```
-- debug_macros.h --
#ifdef DEBUG
#define DEBUG_PRINT(str)    fputs(str, stderr)
#else
#define DEBUG_PRINT(str)    ((void) 0) /* no action */
#endif /* DEBUG */
```

[コード例]

```
void func(void) {
    DEBUG_PRINT(">> func\n");
    ...
    DEBUG_PRINT("<< func\n");
}
```

(b) assertマクロを使用する方法

C言語規格では、プログラム診断機能として、assertマクロが用意されている。assertマクロはデバッグ時にプログラムミスを見つけやすくするために有用である。どのような場所でassertマクロを使用するのかを規定し、プロジェクト内で統一させておくと、結合テストなどの時に一貫したデバッグ情報が収集できデバッグしやすくなる。

以下、assertマクロの使用方法について簡単に説明する。例えば、引数として渡されるポインタにナルポインタが渡されることはないという前提条件のもとに書かれた関数定義では、以下のようにコーディングしておく。

```
void func(int *p) {
    assert(p != NULL);
    *p = INIT_DATA;
    ...
}
```

コンパイル時にNDEBUGマクロが定義された場合、assertマクロは何もしない。一方、NDEBUGマクロが定義されない場合はassertマクロに渡した式が偽の場合、ソースのファイル名と行位置を標準エラーに吐き出したのち異常終了する。マクロ名がDEBUGではなく、NDEBUGであることに注意すること。

assertマクロは、コンパイラが<assert.h>で用意するマクロである。以下を参考に、プロジェクトで異常終了のさせ方を検討し、コンパイラが用意しているマクロを利用するか、独自のassert関数を用意するかを決定する。

```
#ifndef 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();
}
```

C11では、コンパイル時に評価可能な静的アサートをソースコードに埋めておき、構造体メンバのオフセットや文字列定数の長さの確認をコンパイル時に行うことができる。

```
_Static_assert( sizeof(t) <= 4, " tのサイズが4バイトを超えている。");
```

●リリース後のログ出力

デバッグのための記述を含まないリリース用モジュールにも、問題調査のための記述を入れておくとう用である。よくある方法は、調査情報をログとして残すことである。ログ情報は、リリースモジュールに対する妥当性確認のテストや、顧客に提供したシステムで発生した問題の調査に役立つ。

ログ情報を残す場合、次のような項目を決定し、規約として規定する。

・ タイミング

ログを残すタイミングは、異常状態発見時だけでなく、外部システムとの送受信のタイミングなど、どうして異常状態が発生したのかの過去の履歴が追えるようなタイミングを設定する。

・ ログに残す情報

直前に実行していた処理、データの値、メモリのトレース情報など、どうして異常状態が発生したのかの過去の履歴が追えるような情報を残す。

・ 情報出力用マクロ、または関数

ログ情報の出力は、マクロまたは関数として局所化する。ログの出力先は変更できる方が好ましいことが多いためである。



(1) 前処理演算子#と##を使用してはならない。

[MISRA C:2012 R20.10]

(2) # 演算子の直後に続くマクロパラメータの直後に ## 演算子が続けない。[MISRA C:2012 R20.11]

選択指針

規約化

選

適合例

(2) の適合例

```
#define AAA(a, b)  a#b
#define BBB(x, y)  x##y
```

不適合例

(1)、(2) の不適合例

```
#define XXX(a, b, c)  a#b##c
```

演算子と ## 演算子の評価順序は規定されておらず、# 演算子と ## 演算子を混在させたり、2回以上使用してはならない。

M5.13

関数形式のマクロよりも、関数を使用する。

選択指針

規約化

適合例

```
int func(int arg1, int arg2)
{
    return arg1 + arg2;
}
```

不適合例

```
#define func(arg1, arg2) ((arg1 + arg2))
```

関数形式のマクロではなく、関数にすることでデバッグ時に関数の先頭でストップするなど、処理を追いやすくなる。

また、コンパイラによる型チェックが行われるため、コーディングミスを見つけやすくなる。

関数はインライン関数にしてもよい。インライン関数の性能とオブジェクトサイズについては、E1.1.1を参照。

〔関連ルール〕

E1.1.1, P2.1.1

保守性

5.2

動的なメモリ割り当ての使用に気を付ける。

M5.2.1

(1) 動的メモリは使用しない。

(2) 動的メモリを使用する場合は、《使用するメモリ量の上限、メモリ不足の場合の処理、及びデバッグ方法などを規定する。》

選択指針

規約化 選規

動的メモリを使用すると、不当なメモリをアクセスしたり、メモリをシステムに返却し忘れたりすることにより、システム資源がなくなるメモリリークの危険がある。動的メモリを使用せざるを得ない場合は、デバッグしやすくするためのルールを決めておくといよい。

コンパイラによっては、下記のようなデバッグ用の関数が用意されているものもある。まずは使用しているコンパイラを確認する。また、オープンソースの中にもデバッグ用のソースコードがあるので、自作する場合は、これらを参考に、作成するとよい。

ルール定義の例：

動的メモリ獲得・返却は、mallocやfreeなどの標準関数は使用せずに、プロジェクトで用意したX_MALLOC、X_FREE関数を使用する。デバッグ用のコードは-DDEBUGでコンパイルして作成する。

```
-- X_MALLOC.h --
#ifdef DEBUG
void *log_malloc(size_t size, char*, char*);
void log_free(void*);
#define X_MALLOC(size) log_malloc(size, __FILE__, __LINE__)
#define X_FREE(p) log_free(p, __FILE__, __LINE__)
#else
#include <stdlib.h>
#define X_MALLOC(size) malloc(size)
#define X_FREE(p) free(p)
#endif
```

[コード例]

```
#include "X_MALLOC.h"
...
p = X_MALLOC(sizeof(*p) * NUM);
if (p == NULL) {
    return (MEM_NOTHING);
}
...
X_FREE(p);
return (OK);
```

●参考：動的メモリ使用時の問題点

動的メモリを使用する際に起こしがちな問題を以下にあげる。

・バッファオーバーフロー

獲得したメモリの範囲を超えて、参照または更新を行うことである。特に、範囲外を更新してしまった場合、更新した箇所ですら障害が発生する訳ではなく、更新によって破壊されたメモリを参照した箇所ですら障害が発生する。厄介なのは、動的メモリの場合、どこでメモリを破壊しているかを見つけるのが、非常に困難なことである。

・初期化漏れ

通常の動的メモリ関数で獲得したメモリの初期化は行われていない（一部の動的メモリ獲得関数では初期化を行っているものもある）。自動変数と同様に、プログラムで初期化してから使用しなければならない。

・メモリリーク

返却し忘れているメモリのこと。1回1回終わってしまうプログラムの場合は問題ないが、動作し続けるプログラムの場合、メモリリークの存在は、メモリが枯渇し、システム異常が発生する危険の原因となる。

・返却後使用

free 関数などでメモリを返却すると、そのメモリは、その後に呼ばれる malloc 関数などで再利用される可能性がある。このため、free したメモリのアドレスを使って更新した場合、別用途で利用しているメモリを破壊することになる。バッファオーバーフローでも説明したように、この問題は非常にデバッグが困難である。

これらの問題を起こすコードは、コンパイルエラーにならない。また、問題を埋め込んだ場所で障害が発生しないため、通常の仕様を確認するテストでは発見できない。コードレビューや、これらの問題を発見するためのテストコードを挿入したり、特別なライブラリを組み込んだりしてテストを実施しないとデバッグできない。



移植性

組込みソフトウェアの特徴の1つは、それが動作するプラットフォームの選択肢が多様である点があげられます。すなわち、ハードウェア・プラットフォームとしてのMPUの選択やソフトウェア・プラットフォームであるOSの選択など様々な組み合わせが考えられます。そして、組込みソフトウェアで実現する機能の増大とともに、1つのソフトウェアを様々なプラットフォームに対応させる形で、既存のソフトウェアを別のプラットフォームに移植する機会が増えてきています。

こうした中で、ソフトウェアの移植性は、ソースコードレベルでも極めて重要な要素になりつつあります。特に、利用するコンパイラなどに依存する書き方などは日常的にも犯しやすい誤りの1つです。

- 移植性 1 … コンパイラに依存しない書き方にする。
- 移植性 2 … 移植性に問題のあるコードは局所化する。

コンパイラに依存しない書き方にする。

C言語でプログラミングをする以上、コンパイラ（処理系）の利用は避けて通れません。世の中には様々な処理系が提供されていますが、それぞれ若干の癖をもっています。そして、ソースコードを作成する際に下手な書き方をすると、この処理系の癖に依存する形となってしまう、別の処理系を用いた場合に思わぬ事態となります。

このためプログラミングをする際には、処理系に依存しない書き方にするといった注意が必要です。

移植性 1.1

拡張機能や処理系定義の機能は使用しない。

移植性 1.2

言語規格で定義されている文字や拡張表記のみを使用する。

移植性 1.3

データ型の表現、動作仕様の拡張機能、及び処理系依存部分を確認し、文書化する。

移植性 1.4

ソースファイル取込みについて、処理系依存部分を確認し、依存しない書き方にする。

移植性 1.5

コンパイル環境に依存しない書き方にする。

拡張機能や処理系定義の機能は使用しない。

P1.1.1

- (1) 言語標準の規格外の機能は使用しない。
(2) 言語標準の規格外の機能を使用する場合は、
《使用する機能とその使い方を文書化する。》

選択指針	
規約化	選文

現時点で普及しているC言語の標準規格はC99であるが、最新の標準規格はC11である。また、多くのコンパイラは旧規格であるC90に対しても有効である。

(2) を選択して、最新規格C11で定義されコンパイラで部分的にサポートされている機能について利用を認めるというルールも考えられる。

規格外機能の利用方法に関して、以下の関連ルールで詳細を示している。

〔関連ルール〕

P1.1.3, P1.2.1, P1.2.2, P1.3.2, P2.1.1, P2.1.2

P1.1.2

《使用する処理系定義の動作はすべて文書化しなければならない。》[MISRA C:2004 3.1]

選択指針	○
規約化	文

言語規格には動作が処理系によって異なる処理系定義項目がある。例えば、次は処理系定義であり、使用する場合には文書化の対象となる。

- ・浮動小数点の表現方法
- ・C90の場合、整数除算の剰余の符号の扱い
- ・インクルード指令のファイル検索順序
- ・#pragma

P1.1.3

他言語で書かれたプログラムを利用する場合、《そのインタフェースを文書化し、使用方法を規定する。》

選択指針	
規約化	文規

C言語規格では、他の言語で書かれたプログラムをC言語プログラムから利用するためのインタフェースを定めていない。すなわち、他言語で書かれたプログラムを利用する場合は、拡張機能を利用することとなり、移植性に欠ける。使用する場合は、移植の可能性の有無に関わらず、コンパイラの仕様を文書化するとともに使用方法を規定する。

〔関連ルール〕

P1.1.1, P2.1.1

移植性 1.2

言語規格で定義されている文字や拡張表記のみを使用する。

P1.2.1

プログラムの記述において、言語規格で規定している文字以外の文字を使用する場合、コンパイラの仕様を確認し《その使い方を規定する。》

選択指針	
規約化	規

言語規格において、ソースコードで利用できる文字として最低限定義しているのは、アルファベットの大文字と小文字、数字、記号文字（_[]{}#()<>%:;.,?*+ - / ^ & | ~ = , " '）、空白文字、水平タブ、垂直タブ、及び書式送りを表す制御文字である。

識別子や文字に国際文字名や多バイト文字（日本語など）が使用できるが、サポートしていない処理系もあり、それらを使用する場合には、以下の場所に使用できるかを確認し、その使い方を規定する。

- ・ 識別子
- ・ コメント
- ・ 文字列リテラル
 - 文字列の文字コード中に「\」が存在した場合の処理（特別な配慮が必要か、コンパイル時のオプション指定が必要か、など）
 - ワイド文字列リテラル（L “文字列” のようにL という接頭語）で記述する必要性

- ・文字定数
 - その文字定数のビット長
 - ワイド文字定数 (L ‘あ’ のようにL という接頭語) で記述する必要性
- ・#includeのファイル名

例えば、以下の様なルールを規定する。

- ・識別子には英数字と下線だけを使用する。
- ・コメントに日本語を使用してもよい。使用する日本語のコードはShift_JISとする。半角カナは使用しない。
- ・文字列、文字定数、#includeのファイル名には日本語を使用しない。

[関連ルール]

M4.3.1, M4.3.2, P1.1.1



言語規格で定義されているエスケープシーケンス だけを使用する。

選択指針

規約化

適合例

```
char c = '\t'; /* OK */
```

不適合例

```
char c = '\e'; /*NG 言語規格で定義されていない  
エスケープシーケンス。移植性はない。 */
```

言語規格では、非図形文字のエスケープシーケンスとして次の7つが規定されている。

\a (警報)、\b (後退)、\f (書式送り)、\n (改行)、\r (復帰)、\t (水平タブ)、\v (垂直タブ)

[関連ルール]

P1.1.1

データ型の表現、動作仕様の拡張機能、及び処理系依存部分を確認し、文書化する。

P1.3.1

単なる（符号指定のない）char型は、文字の値の格納（処理）にだけ使用し、符号の有無（処理系定義）に依存する処理が必要な場合は、符号を明記したunsigned charまたはsigned charを利用する。

選択指針

○

規約化

適合例

```
char c = 'a';    /* 文字の格納に利用 */
int8_t i8 = -1; /* 8bitのデータとして利用したい
                場合は、例えばtypedefした型を使用する */
```

不適合例

```
char c = -1;
if (c > 0) { ... }
/* 不適合：処理系によってcharは符号付きの場合と
   符号なしの場合があり、その違いで、比較の結果が
   異なる */
```

符号を指定しないcharは、intなどの他の整数型と違い、符号付きか否かがコンパイラによって異なる（intはsigned intと同じ）。このため符号の有無に依存する使用は移植性がない。これは符号指定のないcharが、文字の格納のために用意された独立した型（char、unsigned char、signed charは、3つの型）であり、言語規格はのために利用することを想定しているためである。符号の有無に依存する処理が必要であるなど小さい整数型として使用する場合には、符号を指定したunsigned charまたは、signed charを使用する。その場合、移植時の修正範囲を局所化するため、その型をtypedefして使用することが望ましい。

本ルールと似ている問題に標準関数のgetcの返す型がintであり、charで受けてはいけないというものがあるが、これは関数インタフェース（情報損失の可能性のある代入）の問題である。

【 より詳しく知りたい人への参考文献 】

"MISRA C:2012" Rule 10.1

【 関連ルール 】

P2.1.3

P1.3.2

列挙型 (enum 型) のメンバは、int 型で表現可能な値で定義する。

選択指針

規約化

適合例

```
/* int 16bit、long32bitの場合 */
enum largenum {
    LARGE = INT_MAX
};
```

不適合例

```
/* int 16bit、long32bitの場合 */
enum largenum {
    LARGE = INT_MAX+1
};
```

C 言語規格では、列挙型のメンバは int 型で表現できる範囲の値でなければならない。しかし、コンパイラによっては、機能を拡張し int 型の範囲を超えていてもエラーにならない可能性がある。

参考 C++ では long 型の範囲の値が許されている。

[関連ルール]

P1.1.1

P1.3.3

- (1) ビットフィールドは使用しない。
- (2) ビット位置が意識されたデータに対してはビットフィールドは使用しない。
- (3) ビットフィールドの処理系定義の動作とパッキングに (プログラムが) 依存している場合、《それは文書化しなければならない》[MISRA C:2004 3.5]

選択指針

○

規約化

選文

適合例

```
(2) の適合例
struct S {
    unsigned int bit1:1;
    unsigned int bit2:1;
};
extern struct S * p; /* 例えば p は、単なる
    フラグの集合を指しているなど、p が指すデータ中、
    bit1 がどのビットであってもよい場合は、OK */
p->bit1 = 1;
```

不適合例

```
(2) の不適合例
struct S {
    unsigned int bit1:1;
    unsigned int bit2:1;
};
extern struct S * p; /* 例えば p が IO ポートを
    指しているなどビット位置が意味をもつ、すなわち、
    bit1 がデータの最低位ビットを指すか、最高位ビット
    を指すかに意味がある場合は、移植性がない */

p->bit1 = 1; /* p が指しているデータのどの
    ビットに設定されるかは処理系依存 */
```

移植性

1

ビットフィールドは、次の動作がコンパイラによって異なる。

- (1) 符号指定のないint型のビットフィールドが符号付と扱われるかどうか。
- (2) 単位内のビットフィールドの割り付け順序
- (3) ビットフィールドを記憶域単位の境界

例えばIOポートへのアクセスのように、ビット位置が意味をもつデータのアクセスに使用すると、(2)、(3)の点から移植性に問題がある。そのため、そのような場合にはビットフィールドは利用せず、&や|などのビット単位の演算を使用する。

【関連ルール】

R2.6.1

移植性 1.4

ソースファイル取込みについて、処理系依存部分を確認し、依存しない書き方にする。

P1.4.1

#include 指令の後には、<filename> または "filename" が続かなければならない。

[MISRA C:2012 R.20.3]

選択指針



規約化

適合例

```
#include <stdio.h>
#include "myheader.h"
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h"
#endif
#include INCFILE
```

不適合例

```
#include stdio.h
/* < > も、" " も付いていない */
#include "myheader.h" 1
/* 最後に1が指定されている */
```

C言語規格では、#include 指令をマクロ展開した後の形がヘッダ名の2つの形式 (<...> と "...") のどちらとも一致しない場合の動作は定義されていない (未定義)。ほとんどのコンパイラが、どちらの形式にも一致しない場合をエラーにするが、エラーにしない場合もあるので、いずれかの形式で記述する。

P1.4.2

《#includeのファイル指定で、<>形式と" "形式の使い分け方を規定する。》

選択指針

規約化

規

適合例

```
#include <stdio.h>
#include "myheader.h"
```

不適合例

```
#include "stdio.h"
#include <myheader.h>
```

#includeの書き方には2種類ある。使い方を統一するために例えば以下のようなルールを規定する。

- ・コンパイラの提供するヘッダは、<>で囲んで指定する
- ・プロジェクトで作成したヘッダは、" "で囲んで指定する
- ・購入ソフトの提供するヘッダは、" "で囲んで指定する

P1.4.3

#includeのファイル指定では、文字'、\、"、/*、//、及び:は使用しない。

選択指針

○

規約化

適合例

```
#include "inc/my_header.h" /* OK */
```

不適合例

```
#include "inc¥my_header.h" /* NG */
```

これらの文字を使用する(正確には次に示す)場合、C言語規格では、動作が定義されていない。すなわち、どのような動作となるか不定であり、移植性がない。

- ・文字'、\、"、/*、または//が<>の間の文字列中に現れた場合
- ・文字'、\、/*、または//が"の間の文字列中に現れた場合

その他、文字:(コロン)は処理系により動作が異なるため、移植性がない。

コンパイル環境に依存しない書き方にする。

P15.1

#include のファイル指定では、絶対パスは記述しない。

選択指針

規約化

適合例

#include "h1.h"

不適合例

#include "/project1/module1/h1.h"

- 絶対パスで記述すると、ディレクトリを変更してコンパイルするときに修正が必要となる。

移植性に問題のあるコードは局所化する。

処理系に依存するソースコードは極力書かないようにすることが原則ですが、場合によっては、そのような記述を避けられない場合があります。代表的なものとしては、C言語からアセンブリ言語のプログラムを呼び出す場合などがそれにあたります。このような場合には、その部分をできるだけ局所化しておくことが推奨されます。

移植性 2.1

移植性に問題のあるコードは局所化する。

移植性に問題のあるコードは局所化する。

P2.1.1

C言語からアセンブリ言語のプログラムを呼び出す場合、インラインアセンブリ言語のみが含まれるC言語の関数として表現する、またはマクロで記述するなど、《局所化する方法を規定する。》

選択指針

○

規約化

規

適合例

```
#define SET_PORT1 asm("    st.b 1, port1")
void f() {
    ...
    SET_PORT1;
    ...
}
```

不適合例

```
void f() {
    ...
    asm("    st.b 1,port1");
    ...
}
/* asmと他の処理が混在している */
```

C99ではインライン指定の関数とする。アセンブラを取り込む方式としてasm(文字列)の形式を拡張サポートしている処理系が多い。しかしながら、サポートしていない処理系もあり、また、同じ形式でも違う動作となる場合もあり移植性はない。

[関連ルール]

M5.1.3, P1.1.1, P1.1.3, E1.1.1

P2.1.2

処理系が拡張しているキーワードは、《マクロを規定して》局所化して使用する。

選択指針

○

規約化

規

適合例

```
/* interruptはある処理系が機能拡張している
   キーワードとする。*/
#define INTERRUPT interrupt
INTERRUPT void int_handler(void) {
    ...
}
```


不適合例

```
/* interruptはある処理系が機能拡張している
   キーワードとする。マクロ化せずに利用している */
interrupt void int_handler(void) {
    ...
}
```

コンパイラによっては、#pragma 指令を用いず、拡張したキーワードを提供している場合があるが、そのようなキーワードの使用は移植性がない。使用する場合は、マクロ化するなど局所化することが重要である。マクロ名は、例に示すようにキーワードを大文字にしたものを利用することが多い。

[関連ルール]

P1.1.1



(1) char、int、long、long long、float、double 及び long double という基本型は使用しない。代わりに typedef した型を使用する。《プロジェクトで利用する typedef した型を規定する。》

(2) char、int、long、long long、float、double 及び long double という基本型を、そのサイズに依存する形式で使用する場合、各基本型を typedef した型を使用する。《プロジェクトで利用する typedef 型を規定する。》

選択指針	○
規約化	選規

適合例

(1)、(2) の適合例

```
uint32_t flag32; /* 32bit を仮定する場合、uint32_t を利用 */
```

(2) の適合例

```
int i;
for (i = 0; i < 10; i++) { ... }
/* i はインデックスとして利用。8bit でも、16bit でも、32bit でも OK であり、言語仕様の基本型を利用して OK */
```

不適合例

(1)、(2) の不適合例

```
unsigned int flag32; /* int を 32bit と仮定して利用 */
```

整数型や浮動小数点型のサイズと内部表現はコンパイラによって異なる。

C99 では、言語規格として次の typedef を提供することが規定されているため、この型定義を利用する。

int8_t、int16_t、int32_t、int64_t、uint8_t、uint16_t、uint32_t、uint64_t

C90 のコンパイラを利用する場合は、この名前を参考にするといふ。

[関連ルール]

P1.3.1



効率性

組込みソフトウェアは、製品に組み込まれてハードウェアとともに、実世界で動作する点が特徴です。製品コストをより安価にするためのMPUやメモリなどの様々な制約が、ソフトウェアにも課せられます。

また、リアルタイム性の要求などから、厳しい時間制約をクリアしなければなりません。組込みソフトウェアでは、メモリなどの資源効率性や時間性能を考慮した時間効率性に注意しながらコーディングする必要があります。

- 効率性 1 … 資源や時間の効率を考慮した書き方にする。

効率性

1

資源や時間の効率を考慮した書き方にする。

ソースコードの書き方により、オブジェクトサイズが増加してしまったり、実行速度が低下してしまうことがあります。メモリサイズや処理時間に制約がある場合には、それらを意識したコードの書き方を工夫する必要があります。

効率性 1.1

資源や時間の効率を考慮した書き方にする。

資源や時間の効率を考慮した書き方にする。



E1.1.1

マクロ関数は、速度性能に関わる部分に閉じて使用する。

選択指針

●

規約化

適合例

```
extern void func1(int,int); /* func1:1度しか呼ばれない */
#define func2(arg1, arg2) ... /* func2:何度も呼ばれる */

func1(arg1, arg2);
for (i = 0; i < 10000; i++) {
    func2(arg1, arg2); /* 速度性能が重要な処理 */
}
```

不適合例

```
#define func1(arg1, arg2) ... /* func1:1度しか呼ばれない */
extern void func2(int, int); /* func2:何度も呼ばれる */

func1(arg1, arg2);
for (i = 0; i < 10000; i++) {
    func2(arg1, arg2); /* 速度性能が重要な処理 */
}
```

マクロ関数よりも関数の方が安全であり、なるべく関数を使用する（M5.1.3参照）。しかし、関数は呼出しと復帰の処理で速度性能が劣化する可能性がある。インライン関数の使用が考えられるが、インライン展開は処理系依存で展開されない場合もある。このため速度性能を上げる場合、マクロ関数を使用する。

ただし、マクロ関数を多用すると使用した場所にコードが展開され、オブジェクトサイズが増加する可能性がある。

【関連ルール】

M5.1.3



E1.1.2

繰り返し処理内で、変化のない処理を行わない。

選択指針

●

規約化

適合例

```
var1 = func();
for (i = 0; (i + var1) < MAX; i++) {
    ...
}
```

不適合例

```
/* 関数 func は、同じ結果を返す */
for (i = 0; (i + func()) < MAX; i++) {
    ...
}
```

同じ結果が返される場合に、同じ処理を複数実施すると非効率である。コンパイラの最適化に頼れることも多いが、例のようにコンパイラでは分からない場合は注意する。

E1.1.3

関数の引数として構造体ではなく構造体ポインタを使用する。

選択指針

規約化

適合例

```
typedef struct stag {
    int mem1;
    int mem2;
    ...
} STAG;
int func (const STAG *p) {
    return p->mem1 + p->mem2;
}
```

不適合例

```
typedef struct stag {
    int mem1;
    int mem2;
    ...
} STAG;
int func (STAG x) {
    return x.mem1 + x.mem2;
}
```

- 関数の引数として構造体を渡すと、関数呼出し時に構造体のデータをすべて実引数のための領域にコピーする処理が行われ、構造体のサイズが大きいと、速度性能を劣化させる原因となる。
- 参照し olmayan 構造体を渡す場合は、単に構造体ポインタにするだけでなく、const 修飾を行うこと。

E1.1.4

《switch文とするかif文とするかは、可読性と効率性を考えて選択方針を決定し、規定する。》

選択指針

規約化

規

switch 文は if 文より可読性に優れることが多い。また、最近のコンパイラは、switch 文に対してテーブルジャンプ、バイナリサーチなどの最適化したコードを出力することが多い。このことを考慮してルールを規定する。

ルールの例：

式の値（整数値）によって処理を分岐する場合、分岐の数が3以上であれば、if 文ではなく switch 文を使用する。ただし、プログラムの性能向上において switch 文の効率が問題となる場合には、この限りではない。

Part 3

組込みソフトウェアに ありがちなコーディングミス

1 意味のない式や文

2 誤った式や文

3 誤ったメモリの使用

4 論理演算の勘違いによる誤り

5 タイプミスによる誤り

6 コンパイラによってはエラーにならないケースがある記述

組込みソフトウェアにありがちなコーディングミス

C言語の初心者だけでなく熟練したプログラマでも誤ることのあるコーディングミスの記述の例を示します。最近のコンパイラでは、警告機能を強化したものをオプションで用意している場合があります、ここで説明した幾つかは、コンパイラの警告や静的解析ツールなどで、チェックすることも可能ですが、コーディング段階で、気を付けておくことにより、後工程作業の工数を削減することが期待できます。

世の中にあるコーディング規約の中には、このような記述誤りについても、ルールとして組み入れているものもあります。コーディング規約に入れるかどうかは、開発に携わる人のスキルレベルなども考慮に入れ、検討することを推奨します。

ここでは、次の6つの点について例を挙げて解説します。

- ・ 意味のない式や文
- ・ 誤った式や文
- ・ 誤ったメモリの使用
- ・ 論理演算の勘違いによる誤り
- ・ タイプミスによる誤り
- ・ コンパイラによってはエラーにならないケースがある記述



1 意味のない式や文

ソースコード中に実行されることのない文や式などを記述したままにしておくと誤解をまねきやすく、結果として不具合につながる場合が少なくありません。特に、ソースコードを作成した技術者とは別の技術者が手を加えたりする場合などは、混乱をまねきやすいと言われています。

例1 実行されない文を記述

```
return ret;  
ret = ERROR;
```

プログラムの制御の流れを分岐させる文 (return、continue、break、goto 文) を入れる場所

を間違えたか、これらの分岐文を記述した時に不要な文を削除し忘れたかのどちらかにより発生する問題です。

例2 実行結果が使用されない文を記述

```
void func( ... ) {  
    int    cnt;  
    ...  
    cnt = 0;  
    return;
```

自動変数や仮引数は、関数復帰後は参照できなくなりますので、自動変数や仮引数の更新から return 文までの間で、更新した変数の参照がない場合、その更新は不要な式（文）ということになります。何らかの処理漏れが考えられます。または、プログラム修正時に、不要な文を削除し忘れた可能性もあります。

例3 実行結果が使用されない式を記述

```
int func( ... ) {  
    int    cnt;  
    ...  
    return cnt++;
```

後置の ++ 演算は、変数の値を参照後に更新されるため、例のようなインクリメントには意味がありません。インクリメントした後の値を呼出し元に返したい場合は、前置のインクリメントにしなければなりません。

例4 実引数で渡した値が使用されない

```
int func(int in) {  
    in = 0; /* 引数を上書き */  
    ...  
}
```

例のように、仮引数を参照することなしに上書きしてしまうことは、呼出し元が設定した実引数の値を無視することになります。また、仮引数を変更しても呼出し元の実引数は変更されません。コーディングミス可能性があります。



2 誤った式や文

ソースコードを作成するということは、利用するプログラミング言語で決められた文法にのっとってコードを記述する必要があります。プログラミング言語に精通していても、うっかりミスに近い過ちを犯す場合があります。以下によく見かける誤った式や文を例として示します。

例1 誤った範囲指定

```
if (0 < x < 10)
```

例のプログラムは、一見正しい記述のように感じますが、C言語では、上記のような記述は、数学的な解釈をしてくれません。必ず真となる条件式になってしまいます。

例2 範囲外の比較

```
unsigned char uc;
unsigned int  ui;
...
if (uc == 256)
...
switch (uc) {
case 256:
...
}
if (ui < 0)
...
```

変数が表現可能な範囲を超えた値との比較を行っています。ucは0から255までの値しか表現できません。uiは負になることはありません。

例3 文字列の比較は== 演算では行えない

```
if (str == "abc")
```

例の条件は、アドレスの比較であり、"abc"という文字列が、strの指す文字列と等しいかどうかの条件になっていません。

例4 関数の型と return 文の不整合

```
int func1(int in) {
    if (in < 0) return; /* NG */
    return in ;
}
int func2(void) { /* NG */
    ...
    return;
}
```

値を返す関数の定義では、すべての return 文で、返すべき値を return 式で記述しなければなりません (func1 関数)。また、値を返さない return 文をもつ関数の型は void 型とすべきです (func2 関数)。C99 の場合、このような関数の型と return 文の不整合はコンパイラでエラーとして検出されます。



3 誤ったメモリの使用

C 言語の特徴の 1 つにメモリを直接操作できる点があります。これは組込みソフトウェアを作る上で、大変に有効な長所となる一方で、誤った操作を招く場合も少なくなく、注意が必要です。

例1 配列の範囲外の参照・更新

```
char var1[N];
...
for (i = 1; i <= N; i++) { /* 配列の範囲外をアクセス (NG) */
    var1[i] = i;
}
var1[-1] = 0; /* NG */
var1[N] = 0; /* NG */
```

C 言語における配列のインデックスは 0 から始まり、最大値は要素数より 1 つ少ない値です。

例2 自動変数の領域のアドレスを呼出し元に渡してしまう誤り

```
int *func(struct tag *p) {
    int x;
    p->mem = &x; /* 関数復帰後に自動変数領域が参照されてしまう（危険） */
    return &x; /* 関数復帰後に自動変数領域が参照されてしまう（危険） */
}

...

struct tag y;
int *p;
p = func(&y);
*p = 10; /* 不当な領域を破壊 */
*y.mem = 20; /* 不当な領域を破壊 */
```

自動変数や引数のための領域は、関数が終了するとシステムに解放され、他の用途に再利用される可能性があります。例のように、自動変数領域のアドレスを関数の戻り値に指定したり、呼出し元が参照できる領域に設定してしまうと、システムに返却された領域を参照したり更新できてしまうため、思わぬ障害を発生する危険があります。

C99で導入された複合リテラルのための領域は、それを囲むブロックを抜けると解放され、他の用途に再利用される可能性があります。

```
void f()
{
    ...
    int *p;
    {
        p = (int []){2, 4};
        ...
    }
    x = p[0]; /* 領域解放後に参照の可能性（危険） */
    ...
}
```

例3 動的メモリ解放後のメモリ参照

```
struct stag { /* リスト構造の構造体 */
    struct stag *next;
    ...
};
struct stag *wkp; /* リスト構造のポインタ */
struct stag *top; /* リスト構造の先頭ポインタ */
...
/* リスト構造の構造体を順次解放する処理 */
/* 解放後、for文の3つ目の制御式で、解放済みのポインタにアクセスしているので、NG */
for (wkp = top; wkp != NULL; wkp = wkp-> next) {
    free(wkp);
}
```

malloc 関数などで獲得したメモリは、free 関数でシステムに解放する必要があります。free 関数で解放した領域は、システムで再利用されるため参照してはなりません。

例4 文字列リテラルを書き込む誤り

```
char *s;
s = "abc"; /* 文字列リテラルの領域はROM領域の可能性あり */
s[0] = 'A'; /* 書き込みNG */
```

文字列リテラルは、コンパイラによってはconst領域に割り付けられる可能性があります。文字列リテラルは、書き換えないように、プログラマが注意しなければなりません。

例5 複写サイズの指定誤り

```
#define A 10
#define B 20
char a[A];
char b[B];
...
memcpy(a, b, sizeof(b));
```

配列から配列に複写を行う場合、複写元のサイズで複写してしまうと、複写元サイズが大きい場合、領域破壊を起こしてしまいます。配列から配列の複写を行う場合は、配列のサイズを同じにするのが一番よい方法ですが、複写サイズを複写先のサイズにしておくと、少なくとも領域破壊を防ぐことができます。



4 論理演算の勘違いによる誤り

論理演算子は比較的誤りやすい部分です。特にこれらが利用される場面では、その演算結果によって、その後の処理内容が変わったりする場合も少なくないので注意が必要です。

例1 論理和とするところを論理積とした誤り

```
if (x < 0 && x > 10)
```

論理和とするところを誤って論理積としてしまった例です。C言語では、あり得ない条件を書いてしまってもコンパイルエラーとはなりませんので注意が必要です。

例2 論理積とするところを論理和とした誤り

```
int i, data[10], end = 0;
for (i = 0; i < 10 || !end; i++) {
    data[i] = 設定データ; /* 領域外破壊の危険有り */
    if (終了条件) {
        end = 1;
    }
}
```

配列の要素を順次、参照したり更新したりする繰り返し文の条件として、配列の範囲を超えないようにする条件に異なる条件を加えた場合、これらの条件は論理積でなければなりません。例のように論理和にしてしまうと、配列の領域外をアクセスしてしまう可能性が発生します。

例3 論理演算とするところをビット演算とした誤り

```
if (len1 & len2)
```

これは論理積演算子(&&)を書くべきところに、ビットAND演算子(&)を記述した例です。ビットAND演算子は、条件を論理積するという意味ではありません。正しくプログラムの意味を記述するようにしましょう。

5 タイプミスによる誤り

C言語の演算子の中には、`=` と `==` のように、ちょっとした不注意やタイプミスによって、全く意味合いが変わってしまうものがあります。これらについても十分に注意が必要です。

例1 `==` 演算子を記述するべきところに `=` 演算子を記述

```
if (x = 0)
```

値が等しいかを調べるのは、`=` ではなく `==` を書かなければなりません。今回のようなコーディングミスを防ぐためのルールとして、「真偽を求める式の中で代入演算子を使用しない」というルールもあります。

`a == b;` のように、`=` 演算子を記述するべきところを誤って `==` 演算子を記述してしまう例もありますので注意しましょう。

6 コンパイラによってはエラーにならないケースがある記述

利用するコンパイラには様々な癖があります。コンパイラによっては不適切な書き方であっても、コンパイルの時点でコンパイルエラーとしないものもあり注意が必要です。

例1 同名マクロの多重定義

```
/* AAAを参照する箇所により、展開されるものが異なる */
#define AAA 100
a = AAA; /* 100が代入される */
#define AAA 10
b = AAA; /* 10が代入される */
```

`#define` で定義したマクロ名を、`#undef` せずに再定義してもコンパイルエラーにしないコンパイラがあります。マクロ名が使用されているところにより、展開されるものが異なるのは可読性を低下させます。

例2 const 領域に書き込む誤り

```
void func(const int *p) {  
    *p = 0;    /* const 領域に書き込み (NG) */  
}
```

const 領域を書き換えてもコンパイルエラーにならないコンパイラもあります。const 領域を書き換えないように、プログラマが注意しなければなりません。

付録

付録 A 作法・ルール一覧

付録 B C 言語文法によるルール分類

付録 C 処理系定義の動作について

付録 A 作法・ルール一覧

[信頼性 1] R1 領域は初期化し、大きさに気を付けて使用する。			
作法	ルール		ページ
R1.1 領域は、初期化してから使用する。	R1.1.1	自動変数は宣言時に初期化する。または値を使用する直前に初期値を代入する。	27
	R1.1.2	const型変数は、宣言時に初期化する。	27
R1.2 初期化は過不足ないことがわかるように記述する。	R1.2.1	要素数を指定した配列の初期化では、初期値の数は、指定した要素数と一致させる。	28
	R1.2.2	列挙型 (enum型) のメンバの初期化は、定数を全く指定しない、すべて指定する、または最初のメンバだけを指定する、のいずれかとする。	28
R1.3 ポインタの指す範囲に気を付ける。	R1.3.1	(1) ポインタへの整数の加減算 (++, --も含む) は使用せず、確保した領域への参照・代入は [] を用いる配列形式で行う。 (2) ポインタへの整数の加減算 (++, --も含む) は、ポインタが配列を指している場合だけとし、結果は、配列の範囲内を指すようにする。	29
	R1.3.2	ポインタ同士の減算は、同じ配列の要素を指すポインタにだけ使用する。	30
	R1.3.3	ポインタ同士の大小比較は、同じ配列の要素、または同じ構造体のメンバを指すポインタにだけ使用する。	30
	R1.3.4	restrict型修飾子は使用しない。 【MISRA C:2012 R8.14】	31
[信頼性 2] R2 データは範囲、大きさ、内部表現に気を付けて使用する。			
作法	ルール		ページ
R2.1 内部表現に依存しない比較を行う。	R2.1.1	浮動小数点式は、等価または非等価の比較をしない。	33
	R2.1.2	浮動小数点型変数はループカウンタとして使用しない。	33
	R2.1.3	構造体や共用体の比較に memcmp を使用しない。	34
R2.2 論理値などが区間として定義されている場合、その中の一点 (代表的な実装値) と等しいかどうかで判定を行ってはならない。	R2.2.1	真偽を求める式の中で、真として定義した値と比較しない。	34

[信頼性 2] R2 データは範囲、大きさ、内部表現に気を付けて使用する。			
作法		ルール	ページ
R2.3 データ型を揃えた演算や比較を行う。	R2.3.1	符号なし整数定数式は、結果の型で表現できる範囲内で記述する。	35
	R2.3.2	条件演算子 (?: 演算子) では、論理式は括弧で囲み、戻り値は2つとも同じ型にする。	35
	R2.3.3	ループカウンタとループ継続条件の比較に使用する変数は、同じ型にする。	36
R2.4 演算精度を考慮して記述する。	R2.4.1	演算の型と演算結果の代入先の型が異なる場合は、期待する演算精度の型へキャストしてから演算する。	36
	R2.4.2	符号付きの式と符号なしの式の混在した算術演算、比較を行う場合は、期待する型に明示的にキャストする。	37
R2.5 情報損失の危険のある演算は使用しない。	R2.5.1	情報損失を起こす可能性のあるデータ型への代入 (= 演算、関数呼出しの実引数渡し、関数復帰) や演算を行う場合は、問題がないことを確認し、問題がないことを明示するためにキャストを記述する。	38
	R2.5.2	単項演算子 "-" は符号なしの式に使用しない。	39
	R2.5.3	unsigned char 型、または unsigned short 型のデータをビット反転 (~)、もしくは左シフト (<<) する場合、結果の型に明示的にキャストする。	39
	R2.5.4	シフト演算子の右辺の項はゼロ以上、左辺の項のビット幅未満でなければならない。	40
R2.6 対象データが表現可能な型を使用する。	R2.6.1	(1) ビットフィールドに使用する型は signed int と unsigned int だけとし、1ビット幅のビットフィールドが必要な場合は signed int 型でなく、unsigned int 型を使用する。 (2) ビットフィールドに使用する型は signed int、unsigned int、または _Bool とし、1ビット幅のビットフィールドが必要な場合は、unsigned int 型 または _Bool 型を使用する。 (3) ビットフィールドに使用する型は signed int、unsigned int、_Bool または、処理系が許容している型のうち signed と unsigned を指定した型または enum 型を使用する。1ビット幅のビットフィールドが必要な場合は、unsigned を指定した型または _Bool 型を使用する。	41
	R2.6.2	ビット列として使用するデータは、符号付き型ではなく、符号なし型で定義する。	42

【信頼性2】 R2 データは範囲、大きさ、内部表現に気を付けて使用する。			
作法	ルール		ページ
R2.7 ポインタの型に気を付ける。	R2.7.1	(1) ポインタ型は、他のポインタ型及び整数型と相互に変換してはならない。ただし、データへのポインタ型における void* 型との相互変換は除く。 (2) ポインタ型は、他のポインタ型、及びポインタ型のデータ幅未満の整数型と相互に変換してはならない。ただし、データへのポインタ型における void* 型との相互変換は除く。 (3) ポインタ型は、他のポインタ型、及びポインタ型のデータ幅未満の整数型と相互に変換してはならない。ただし、データへのポインタ型における、他のデータへのポインタ型及び void* 型との相互変換は除く。	43
	R2.7.2	ポインタで指し示された型から const 修飾や volatile 修飾を取り除くキャストを行ってはならない。 【MISRA C:2012 R11.8】	45
	R2.7.3	ポインタが負かどうかの比較をしない。	45
R2.8 宣言、使用、定義に矛盾がないことをコンパイラがチェックできる書き方にする。	R2.8.1	引数をもたない関数は、引数の型を void として宣言する。	46
	R2.8.2	(1) 可変個引数をもつ関数を定義してはならない。 【MISRA C:2004 16.1】 (2) 可変個引数をもつ関数を使用する場合は、《処理系での動作を文書化し、使用する》。	46
	R2.8.3	1つのプロトタイプ宣言は1箇所に記述し、それが関数呼出し及び関数定義の両方から参照されるようにする。	47
【信頼性3】 R3 動作が保証された書き方にする。			
作法	ルール		ページ
R3.1 領域の大きさを意識した書き方にする。	R3.1.1	(1) 配列の extern 宣言の要素数は必ず指定する。 (2) 要素数が省略された初期化付き配列定義に対応した配列の extern 宣言を除き配列の extern 宣言の要素数は必ず指定する。	49
	R3.1.2	配列を順次にアクセスするループの継続条件には、配列の範囲内であるかの判定を入れる。	50
	R3.1.3	指示付きの初期化子で初期化する配列のサイズは明示する。	50
	R3.1.4	可変長配列型は使用しない。 【MISRA C:2012 R18.8】	51

【信頼性3】 R3 動作が保証された書き方にする。			
作法		ルール	ページ
R3.2 実行時にエラーになる可能性のある演算に対しては、エラーケースを迂回させる。	R3.2.1	除算や剰余算の右辺式は、0でないことを確認してから演算を行う。	52
	R3.2.2	ポインタは、ナルポインタでないことを確認してからポインタの指す先を参照する。	52
R3.3 関数呼出しではインターフェースの制約をチェックする。	R3.3.1	関数がエラー情報を戻す場合、エラー情報をテストしなければならない。 【MISRA C:2012 D4.7】	53
	R3.3.2	関数は、処理の開始前に引数の制約をチェックする。	54
R3.4 再帰呼出しは行わない。	R3.4.1	関数は、直接的か間接的に関わらず、その関数自身を呼び出してはならない。 【MISRA C:2012 R17.2】	55
R3.5 分岐の条件に気を付け、所定の条件以外が発生した場合の処理を記述する。	R3.5.1	if-else if文は、最後にelse節を置く。通常、else条件が発生しないことが分かっている場合は、次のいずれかの記述とする。 《(i) else節には、例外発生時の処理を記述する。 (ii) else節には、プロジェクトで規定したコメントを入れる。》	56
	R3.5.2	switch文は、最後にdefault節を置く。 通常、default条件が発生しないことが分かっている場合は、次のいずれかの記述とする。 《(i) default節には、例外発生時の処理を記述する。 (ii) default節には、プロジェクトで規定したコメントを入れる。》	57
	R3.5.3	ループカウンタの比較に等価演算子(==、!=)は使用しない。	58
R3.6 評価順序に気を付ける。	R3.6.1	変数の値を変更する記述をした同じ式内で、その変数を参照、変更しない。	59
	R3.6.2	実引数並び、及び2項演算式に、副作用をもつ関数呼出し、volatile変数を、複数記述しない。	60
	R3.6.3	sizeof演算子は、副作用がある式に用いてはならない。	61

【保守性1】 M1 他人が読むことを意識する。			
作法	ルール		ページ
M1.1 使用しない記述を残さない。	M1.1.1	使用しない関数、変数、引数、typedef、タグ、ラベル、マクロなどは宣言（定義）しない。	65
	M1.1.2	コードの一部を"コメントアウト"すべきでない。 【MISRA C:2012 D4.4】	65
M1.2 紛らわしい書き方をしない。	M1.2.1	(1) 1つの宣言文で宣言する変数は1つとする（複数宣言しない）。 (2) 同じような目的で使用する同じ型の自動変数は、1つの宣言文で複数宣言してもよいが、初期化する変数と初期化をしない変数を混在させてはならない。	66
	M1.2.2	適切な型を示す接尾語が使用できる定数記述には、接尾語をつけて記述する。long型整数定数を示す接尾語は大文字の"L"のみ使用する。	67
	M1.2.3	長い文字列リテラルを表現する場合には、文字列リテラル内で改行を使用せず、連続した文字列リテラルの連結を使用する。	67
M1.3 特殊な書き方はしない。	M1.3.1	switch（式）の式には、真偽結果を求める式を記述しない。	68
	M1.3.2	switch文のcaseラベル及びdefaultラベルは、switch文本体の複文（その中に入れ子になった複文は除く）にのみ記述する。	68
	M1.3.3	関数や変数の定義や宣言では型を明示的に記述する。	69
M1.4 演算の実行順序が分かりやすいように記述する。	M1.4.1	&&演算や 演算の右式と左式は二項演算を含まない式か()で囲まれた式を記述する。ただし、&&演算が連続して結合している場合や、 演算が連続して結合している場合は、&&式や 式を()で囲む必要はない。	69
	M1.4.2	《演算の優先順位を明示するための括弧の付け方を規定する。》	70
M1.5 省略すると誤解をまねきやすい演算は、明示的に記述する。	M1.5.1	関数識別子（関数名）には、前に&を付けるか、括弧付きの仮引数リスト（空でも可）を指定して使用しなければならない。 【MISRA C:2004 16.9】	70
	M1.5.2	条件判定の式では、0との比較は明示的にする。	71

【保守性1】 M1 他人が読むことを意識する。			
作法		ルール	ページ
M1.6 領域は1つの利用目的に使用する。	M1.6.1	目的ごとに変数を用意する。	71
	M1.6.2	(1) 共用体は使用してはならない。 【MISRA C:2004 18.4】 (2) 共用体を使用する場合は、書き込んだメンバで参照する。	72
M1.7 名前を再使用しない。	M1.7.1	名前の一意性は、次の規則に従う。 1. 内側のスコープで宣言された識別子は外側のスコープで宣言された識別子を隠してはならない。 【MISRA C:2012 R5.3】 2. typedef 名は一意な識別子でなければならない。 【MISRA C:2012 R5.6】 3. タグ名は一意な識別子でなければならない。 【MISRA C:2012 R5.7】 4. 外部結合をもつオブジェクトや関数を定義する識別子は一意でなければならない。 【MISRA C:2012 R5.8】 5. 内部結合をもつオブジェクトや関数を定義する識別子は一意にするべきである。 【MISRA C:2012 R5.9】 6. 構造体及び共用体のメンバ名を除いて、あるネームスペースの識別子を、他のネームスペースの識別子と同じ綴りにしてはいけない。 【MISRA C:2004 5.6】	73
	M1.7.2	標準ライブラリの関数名、変数名及びマクロ名は再定義・再利用してはならない。また定義を解除してはならない。	74
	M1.7.3	下線で始まる名前(変数)は定義しない。	75
M1.8 勘違いしやすい言語仕様を使用しない。	M1.8.1	論理演算子 && または の右側のオペランドには、副作用があってはならない。 【MISRA C:2012 R13.5】	76
	M1.8.2	Cマクロは、波括弧で囲まれた初期化子、定数、括弧で囲まれた式、型修飾子、記憶域クラス指定子、do-while-zero 構造にのみ展開されなければならない。 【MISRA C:2004 19.4】	76
	M1.8.3	#line は、ツールによる自動生成以外では使用しない。	77

【保守性1】 M1 他人が読むことを意識する。			
作法	ルール		ページ
M1.8 勘違いしやすい言語仕様を使用しない。	M1.8.4	?? で始まる3文字以上の文字の並び、及び代替される字句表記は使用しない。	77
	M1.8.5	0 で始まる長さ2以上の数字だけの列を定数として使用しない。	78
M1.9 特殊な書き方は意図を明示する。	M1.9.1	意図的に何もしない文を記述しなければならない場合はコメント、空になるマクロなどを利用し、目立たせる。	78
	M1.9.2	《無限ループの書き方を規定する。》	79
M1.10 マジックナンバーを埋め込まない。	M1.10.1	意味のある定数はマクロとして定義して使用する。	79
M1.11 領域の属性は明示する。	M1.11.1	参照しからない領域はconstであることを示す宣言を行う。	80
	M1.11.2	他の実行単位により更新される可能性のある領域はvolatileであることを示す宣言を行う。	81
	M1.11.3	《ROM化するための変数宣言、定義のルールを規定する。》	81
M1.12 コンパイルされない文でも正しい記述を行う。	M1.12.1	プリプロセッサが削除する部分でも正しい記述を行う。	82
【保守性2】 M2 修正誤りのないような書き方にする。			
作法	ルール		ページ
M2.1 構造化されたデータやブロックは、まとまりを明確化する。	M2.1.1	配列や構造体を0以外で初期化する場合は、構造を示し、それに合わせるために波括弧 "{" を使用しなければならない。また、すべて0以外の場合を除き、データは漏れなく記述する。	84
	M2.1.2	if、else if、else、while、do、for、switch 文の本体はブロック化する。	84
M2.2 アクセス範囲や関連するデータは局所化する。	M2.2.1	1つの関数内でのみ使用する変数は関数内で変数宣言する。	85
	M2.2.2	同一ファイル内で定義された複数の関数からアクセスされる変数は、ファイルスコープでstatic変数宣言する。	86
	M2.2.3	同じファイルで定義した関数からのみ呼ばれる関数は、static関数とする。	86
	M2.2.4	関連する定数を定義するときは、#define よりenumを使用する。	87

【保守性3】 M3 プログラムはシンプルに書く			
作法		ルール	ページ
M3.1 構造化プログラミングを行う。	M3.1.1	繰返し文を終了させるために使用する break 文または goto 文は、1 つまでとする。 【MISRA C:2012 R15.4】	89
	M3.1.2	(1) goto 文を使用しない。 (2) goto 文を使用する場合、飛び先は同じブロック、または goto 文を囲むブロック内で、かつ goto 文の後方に宣言されているラベルとする。	90
	M3.1.4	(1) switch 文の case 節、default 節は、必ず break 文で終了させる。 (2) switch 文の case 節、default 節を break 文で終了させない場合は、《プロジェクトでコメントを規定し》そのコメントを挿入する。	91
	M3.1.5	(1) 関数は、1 つの return 文で終了させる。 (2) 処理の途中で復帰する return 文は、異常復帰の場合のみとする。	92
M3.2 1 つの文で 1 つの副作用とする。	M3.2.1	(1) コンマ式は使用しない。 (2) コンマ式は for 文の初期化式や更新式以外では使用しない。	92
	M3.2.2	1 つの文に、代入を複数記述しない。ただし、同じ値を複数の変数に代入する場合を除く。	93
M3.3 目的の違う式は、分離して記述する。	M3.3.1	for 文の 3 つの式には、ループ制御に関わるもののみを記述しなければならない。 【MISRA C:2004 13.5】	93
	M3.3.2	for ループの中で繰返しカウンタとして用いる数値変数は、ループの本体内で変更してはならない。 【MISRA C:2004 13.6】	94
	M3.3.3	(1) 真偽を求める式の中で代入演算子を使用しない。 (2) 真偽を求める式の中で代入演算子を使用しない。ただし慣習的に使う表現は除く。	94
M3.4 複雑なポインタ演算は使用しない。	M3.4.1	3 段階以上のポインタ指定は使用しない。	95

【保守性4】 M4 統一した書き方にする。			
作法	ルール		ページ
M4.1 コーディングスタイルを統一する。	M4.1.1	《波括弧 ({ }) や字下げ、空白の入れ方などのスタイルに関する規約を規定する。》	97
M4.2 コメントの書き方を統一する。	M4.2.1	《ファイルヘッダコメント、関数ヘッダコメント、行末コメント、ブロックコメント、コピーライトなどの書き方に関する規約を規定する。》	100
M4.3 名前の付け方を統一する。	M4.3.1	《外部変数、内部変数などの命名に関する規約を規定する。》	102
	M4.3.2	《ファイル名の命名に関する規約を規定する。》	102
M4.4 ファイル内の記述内容と記述順序を統一する。	M4.4.1	《ヘッダファイルに記述する内容（宣言、定義など）とその記述順序を規定する。》	104
	M4.4.2	《ソースファイルに記述する内容（宣言、定義など）とその記述順序を規定する。》	105
	M4.4.3	外部変数や関数（ファイル内でのみ使用する関数を除く）を使用したり定義する場合、宣言を記述したヘッダファイルをインクルードする。	106
	M4.4.4	外部変数は、複数箇所ですべてで定義しない。	106
	M4.4.5	ヘッダファイルには、変数定義や関数定義を記述しない。	107
	M4.4.6	ヘッダファイルは重複取り込みに耐えうる作りとする。《そのための記述方法を規定する。》	107
M4.5 宣言の書き方を統一する。	M4.5.1	(1) 関数プロトタイプ宣言では、すべての引数に名前を付けない（型だけとする）。 (2) 関数プロトタイプ宣言では、すべての引数に名前を付ける。さらに、引数の型と名前、及び戻り型は、関数定義と文字通りに同じにする。	108
	M4.5.2	構造体タグの宣言と変数の宣言は別々に行う。	109
	M4.5.3	(1) 構造体・共用体・配列の初期値式のリスト、及び列挙子リストの最後の "}" の前に "," を記述しない。 (2) 構造体・共用体・配列の初期値式のリスト、及び列挙子リストの最後の "}" の前に "," を記述しない。ただし、配列の初期化の初期値リストの最後の "}" の前に "," を書くことは許す。	109
M4.6 ナルポインタの書き方を統一する。	M4.6.1	(1) ナルポインタには 0 を使用する。NULL はいかなる場合にも使用しない。 (2) ナルポインタには NULL を使用する。NULL はナルポインタ以外に使用しない。	110

【保守性4】 M4 統一した書き方にする。			
作法	ルール		ページ
M4.7 前処理指令の書き方を統一する。	M4.7.1	演算子を含むマクロは、マクロ本体とマクロ引数を括弧で囲む。	111
	M4.7.2	#ifdef、#ifndef、#ifに対応する#else、#elif、#endifは、同一ファイル内に記述し、《プロジェクトで規定したコメントを入れ対応関係を明確にする。》	111
	M4.7.3	#ifや#elifで、マクロ名が定義済みかを調べる場合は、defined（マクロ名）またはdefined マクロ名により定義済みかを調べる。	112
	M4.7.5	マクロは、ブロック内で#define、または#undefしてはならない。 【MISRA C:2004 19.5】	112
	M4.7.6	#undefは使用してはならない。 【MISRA C:2012 R20.5】	113
	M4.7.7	#ifまたは#elif前処理指令の制御式は、0または1に評価されなければならない。 【MISRA C:2012 R20.8】	114
【保守性5】 M5 試験しやすい書き方にする。			
作法	ルール		ページ
M5.1 問題発生時の原因を調査しやすい書き方にする。	M5.1.1	《デバッグオプション設定時のコーディング方法と、リリースモジュールにログを残すためのコーディング方法を規定する。》	116
	M5.1.2	(1) 前処理演算子#と##を使用してはならない。 【MISRA C:2012 R20.10】 (2) #演算子の直後に続くマクロパラメータの直後に##演算子を続けない。 【MISRA C:2012 R20.11】	118
	M5.1.3	関数形式のマクロよりも、関数を使用する。	119
M5.2 動的なメモリ割り当ての使用に気を付ける。	M5.2.1	(1) 動的メモリは使用しない。 (2) 動的メモリを使用する場合は、《使用するメモリ量の上限、メモリ不足の場合の処理、及びデバッグ方法などを規定する。》	119

[移植性1] P1 コンパイラに依存しない書き方にする。			
作法	ルール		ページ
P1.1 拡張機能や処理系定義の機能は使用しない。	P1.1.1	(1) 言語標準の規格外の機能は使用しない。 (2) 言語標準の規格外の機能を使用する場合は、《使用する機能とその使い方を文書化する。》	123
	P1.1.2	《使用する処理系定義の動作はすべて文書化しなければならない。》 【MISRA C:2004 3.1】	123
	P1.1.3	他言語で書かれたプログラムを利用する場合、《そのインタフェースを文書化し、使用方法を規定する。》	124
P1.2 言語規格で定義されている文字や拡張表記のみを使用する。	P1.2.1	プログラムの記述において、言語規格で規定している文字以外の文字を使用する場合、コンパイラの仕様を確認し《その使い方を規定する。》	124
	P1.2.2	言語規格で定義されているエスケープシーケンスだけを使用する。	125
P1.3 データ型の表現、動作仕様の拡張機能、及び処理系依存部分を確認し、文書化する。	P1.3.1	単なる（符号指定のない）char型は、文字の値の格納（処理）にだけ使用し、符号の有無（処理系定義）に依存する処理が必要な場合は、符号を明記した unsigned char または signed char を利用する。	126
	P1.3.2	列挙（enum）型のメンバは、int型で表現可能な値で定義する。	127
	P1.3.3	(1) ビットフィールドは使用しない。 (2) ビット位置が意識されたデータに対してはビットフィールドは使用しない。 (3) ビットフィールドの処理系定義の動作とパッキングに（プログラムが）依存している場合、《それは文書化しなければならない》 【MISRA C:2004 3.5】	127
P1.4 ソースファイル取込みについて、処理系依存部分を確認し、依存しない書き方にする。	P1.4.1	#include 指令の後には、<filename> または "filename" が続くなければならない。 【MISRA C:2012 R20.3】	128
	P1.4.2	《#include のファイル指定で、<> 形式と " " 形式の使い分け方を規定する。》	129
	P1.4.3	#include のファイル指定では、文字 '\', '\n', '/*', '//, 及び ; は使用しない。	129
P1.5 コンパイル環境に依存しない書き方にする。	P1.5.1	#include のファイル指定では、絶対パスは記述しない。	130

[移植性2] P2 移植性に問題のあるコードは局所化する。			
作法		ルール	ページ
P2.1 移植性に問題のあるコードは局所化する。	P2.1.1	C言語からアセンブリ言語のプログラムを呼び出す場合、インラインアセンブリ言語のみが含まれるC言語の関数として表現する、またはマクロで記述するなど、《局所化する方法を規定する。》	132
	P2.1.2	処理系が拡張しているキーワードは、《マクロを規定して》局所化して使用する。	132
	P2.1.3	(1) char、int、long、long long、float、double 及び long double という基本型は使用しない。代わりに typedef した型を使用する。《プロジェクトで利用する typedef した型を規定する。》 (2) char、int、long、long long、float、double 及び long double という基本型を、そのサイズに依存する形式で使用する場合、各基本型を typedef した型を使用する。《プロジェクトで利用する typedef 型を規定する。》	133
[効率性1] E1 資源や時間の効率を考慮した書き方にする。			
作法		ルール	ページ
E1.1 資源や時間の効率を考慮した書き方にする。	E1.1.1	マクロ関数は、速度性能に関わる部分に閉じて使用する。	137
	E1.1.2	繰り返し処理内で、変化のない処理を行わない。	137
	E1.1.3	関数の引数として構造体ではなく構造体ポインタを使用する。	138
	E1.1.4	《switch文とするかif文とするかは、可読性と効率性を考えて選択方針を決定し、規定する。》	138

付録 B C 言語文法によるルール分類

C 言語の文法による分類を示す。

文法による分類	No.	ルール
1 スタイル		
1.1 構文スタイル	M4.1.1	《波括弧 ({ }) や字下げ、空白の入れ方などのスタイルに関する規約を規定する。》
1.2 コメント	M4.2.1	《ファイルヘッダコメント、関数ヘッダコメント、行末コメント、ブロックコメント、コピーライトなどの書き方に関する規約を規定する。》
1.3 名前付け	M1.7.1	<p>名前の一意性は、次の規則に従う。</p> <ol style="list-style-type: none"> 1. 内側のスコープで宣言された識別子は外側のスコープで宣言された識別子を隠してはならない。【MISRA C:2012 R5.3】 2. typedef 名は一意的な識別子でなければならない。【MISRA C:2012 R5.6】 3. タグ名は一意的な識別子でなければならない。【MISRA C:2012 R5.7】 4. 外部結合をもつオブジェクトや関数を定義する識別子は一意的でなければならない。【MISRA C:2012 R5.8】 5. 内部結合をもつオブジェクトや関数を定義する識別子は一意的にするべきである。【MISRA C:2012 R5.9】 6. 構造体及び共用体のメンバ名を除いて、あるネームスペースの識別子を、他のネームスペースの識別子と同じ綴りにしてはいけい。【MISRA C:2004 5.6】
	M1.7.2	標準ライブラリの関数名、変数名及びマクロ名は再定義・再利用してはならない、また定義を解除してはならない。
	M1.7.3	下線で始まる名前 (変数) は定義しない。
	M4.3.1	《外部変数、内部変数などの命名に関する規約を規定する。》
	M4.3.2	《ファイル名の命名に関する規約を規定する。》
1.4 ファイル内の構成	M4.4.1	《ヘッダファイルに記述する内容 (宣言、定義など) とその記述順序を規定する。》
	M4.4.2	《ソースファイルに記述する内容 (宣言、定義など) とその記述順序を規定する。》
	M4.4.3	外部変数や関数 (ファイル内でのみ使用する関数を除く) を使用したり定義する場合、宣言を記述したヘッダファイルをインクルードする。
	M4.4.5	ヘッダファイルには、変数定義や関数定義を記述しない。
	M4.4.6	ヘッダファイルは重複取り込みに耐えうる作りとする。《そのための記述方法を規定する。》

文法による分類	No.	ルール
1 スタイル		
1.5 定数	M1.2.2	適切な型を示す接尾語が使用できる定数記述には、接尾語をつけて記述する。 long 型整数定数を示す接尾語は大文字の "L" のみ使用する。
	M1.2.3	長い文字列リテラルを表現する場合には、文字列リテラル内で改行を使用せず、連続した文字列リテラルの連結を使用する。
	M1.8.5	0 で始まる長さ 2 以上の数字だけの列を定数として使用しない。
	M1.10.1	意味のある定数はマクロとして定義して使用する。
1.6 その他 (スタイル)	M1.1.2	コードの一部を "コメントアウト" すべきでない。 【MISRA C:2012 D4.4】
	M1.8.4	?? で始まる 3 文字以上の文字の並び、及び代替される字句表記は使用しない。
	M1.9.1	意図的に何もしない文を記述しなければいけない場合はコメント、空になるマクロなどを利用し、目立たせる。
	M5.1.3	関数形式のマクロよりも、関数を使用する。
2 型		
2.1 基本型	R2.6.2	ビット列として使用するデータは、符号付き型ではなく、符号なし型で定義する。
	P1.3.1	単なる (符号指定のない) char 型は、文字の値の格納 (処理) にだけ使用し、符号の有無 (処理系定義) に依存する処理が必要な場合は、符号を明記した unsigned char または signed char を利用する。
	P2.1.3	(1) char, int, long, long long, float, double 及び long double という基本型は使用しない。代わりに typedef した型を使用する。《プロジェクトで利用する typedef した型を規定する。》 (2) char, int, long, long long, float, double 及び long double という基本型を、そのサイズに依存する形式で使用する場合、各基本型を typedef した型を使用する。《プロジェクトで利用する typedef 型を規定する。》
2.2 構造体・共用体	R2.1.3	構造体や共用体の比較に memcmp を使用しない。
	M1.6.2	(1) 共用体は使用してはならない。【MISRA C:2004 18.4】 (2) 共用体を使用する場合は、書き込んだメンバで参照する。
	M1.7.2	標準ライブラリの関数名、変数名及びマクロ名は再定義・再利用してはならない、また定義を解除してはならない。
	M4.5.2	構造体タグの宣言と変数の宣言は別々に行う。
2.3 ビットフィールド	R2.6.1	(1) ビットフィールドに使用する型は signed int と unsigned int だけとし、1 ビット幅のビットフィールドが必要な場合は signed int 型でなく、unsigned int 型を使用する。 (2) ビットフィールドに使用する型は signed int、unsigned int、または _Bool とし、1 ビット幅のビットフィールドが必要な場合は、unsigned int 型または _Bool 型を使用する。 (3) ビットフィールドに使用する型は signed int、unsigned int、_Bool または、処理系が許容している型のうち signed と unsigned を指定した型または enum 型を使用する。1 ビット幅のビットフィールドが必要な場合は、unsigned を指定した型または _Bool 型を使用する。

文法による分類	No.	ルール
2 型		
2.3 ビットフィールド	P1.3.3	<p>(1) ビットフィールドは使用しない。</p> <p>(2) ビット位置が意識されたデータに対してはビットフィールドは使用しない。</p> <p>(3) ビットフィールドの処理系定義の動作とパッキングに（プログラムが）依存している場合、《それは文書化しなければならない》 【MISRA C:2004 3.5】</p>
2.4 列挙型	R1.2.2	列挙型（enum 型）のメンバの初期化は、定数を全く指定しない、すべて指定する、または最初のメンバだけを指定する、のいずれかとする。
	M2.2.4	関連する定数を定義するときは、#define より enum を使用する。
	P1.3.2	列挙（enum）型のメンバは、int 型で表現可能な値で定義する。
3 宣言・定義		
3.1 初期化	R1.1.1	自動変数は宣言時に初期化する。または値を使用する直前に初期値を代入する。
	R1.1.2	const 型変数は、宣言時に初期化する。
	R1.2.1	要素数を指定した配列の初期化では、初期値の数は、指定した要素数と一致させる。
	R3.1.3	指示付きの初期化子で初期化する配列のサイズは明示する。
	M2.1.1	配列や構造体を 0 以外で初期化する場合は、構造を示し、それに合わせるために波括弧 "{" を使用しなければならない。また、すべて 0 以外の場合を除き、データは漏れなく記述する。
	M4.5.3	<p>(1) 構造体・共用体・配列の初期値式のリスト、及び列挙子リストの最後の "}" の前に ";" を記述しない。</p> <p>(2) 構造体・共用体・配列の初期値式のリスト、及び列挙子リストの最後の "}" の前に ";" を記述しない。ただし、配列の初期化の初期値リストの最後の "}" の前に ";" を書くことは許す。</p>
3.2 変数宣言・定義	M1.2.1	<p>(1) 1 つの宣言文で宣言する変数は 1 つとする（複数宣言しない）。</p> <p>(2) 同じような目的で使用する同じ型の自動変数は、1 つの宣言文で複数宣言してもよいが、初期化する変数と初期化をしない変数を混在させてはならない。</p>
	M1.6.1	目的ごとに変数を用意する。
	M1.11.1	参照しかない領域は const であることを示す宣言を行う。
	M1.11.2	他の実行単位により更新される可能性のある領域は volatile であることを示す宣言を行う。

文法による分類	No.	ルール
3 宣言・定義		
3.2 変数宣言・定義	M1.11.3	《ROM化するための変数宣言、定義のルールを規定する。》
	M2.2.1	1つの関数内でのみ使用する変数は関数内で変数宣言する。
	M2.2.2	同一ファイル内で定義された複数の関数からアクセスされる変数は、ファイルスコープでstatic変数宣言する。
	M4.4.4	外部変数は、複数箇所でも定義しない。
3.3 関数宣言・定義	R2.8.1	引数をもたない関数は、引数の型をvoidとして宣言する。
	R2.8.2	(1) 可変個引数をもつ関数を定義してはならない。【MISRA C:2004 16.1】 (2) 可変個引数をもつ関数を使用する場合は、《処理系での動作を文書化し、使用する》
	R2.8.3	1つのプロトタイプ宣言は1箇所に記述し、それが関数呼出し及び関数定義の両方から参照されるようにする。
	M2.2.3	同じファイルで定義した関数からのみ呼ばれる関数は、static関数とする。
	M4.5.1	(1) 関数プロトタイプ宣言では、すべての引数に名前を付けない(型だけとする)。 (2) 関数プロトタイプ宣言では、すべての引数に名前を付ける。さらに、引数の型と名前、及び戻り型は、関数定義と文字通りに同じにする。
3.4 配列宣言・定義	R3.1.1	(1) 配列のextern宣言の要素数は必ず指定する。 (2) 要素数が省略された初期化付き配列定義に対応した配列のextern宣言を除き配列のextern宣言の要素数は必ず指定する。
	R3.1.4	可変長配列型は使用しない。【MISRA C:2012 R18.8】
3.5 その他(宣言・定義)	M1.1.1	使用しない関数、変数、引数、typedef、タグ、ラベル、マクロなどは宣言(定義)しない。
	M1.3.3	関数や変数の定義や宣言では型を明示的に記述する。
4 式		
4.1 関数呼出し	R3.3.1	関数がエラー情報を戻す場合、エラー情報をテストしなければならない。 【MISRA C:2012 D4.7】
	R3.3.2	関数は、処理の開始前に引数の制約をチェックする。
	R3.4.1	関数は、直接的か間接的に関わらず、その関数自身を呼び出してはならない。 【MISRA C:2012 R17.2】
4.2 ポインタ	R1.3.1	(1) ポインタへの整数の加減算(++、--も含む)は使用せず、確保した領域への参照・代入は[]を用いる配列形式で行う。 (2) ポインタへの整数の加減算(++、--も含む)は、ポインタが配列を指している場合だけとし、結果は、配列の範囲内を指すようにする。
	R1.3.2	ポインタ同士の減算は、同じ配列の要素を指すポインタにだけ使用する。

文法による分類	No.	ルール
4 式		
4.2 ポインタ	R1.3.3	ポインタ同士の大小比較は、同じ配列の要素、または同じ構造体のメンバを指すポインタにだけ使用する。
	R2.7.1	(1) ポインタ型は、他のポインタ型及び整数型と相互に変換してはならない。ただし、データへのポインタ型における void* 型との相互変換は除く。 (2) ポインタ型は、他のポインタ型、及びポインタ型のデータ幅未満の整数型と相互に変換してはならない。ただし、データへのポインタ型における void* 型との相互変換は除く。 (3) ポインタ型は、他のポインタ型、及びポインタ型のデータ幅未満の整数型と相互に変換してはならない。ただし、データへのポインタ型における、他のデータへのポインタ型及び void* 型との相互変換は除く。
	R2.7.3	ポインタが負かどうかの比較をしない。
	R3.2.2	ポインタは、ナルポインタでないことを確認してからポインタの指す先を参照する。
	M3.4.1	3段階以上のポインタ指定は使用しない。
	M4.6.1	(1) ナルポインタには 0 を使用する。NULL はいかなる場合にも使用しない。 (2) ナルポインタには NULL を使用する。NULL はナルポインタ以外に使用しない。
4.3 キャスト	R2.4.2	符号付きの式と符号なしの式の混在した算術演算、比較を行う場合は、期待する型に明示的にキャストする。
	R2.7.2	ポインタで指し示された型から const 修飾や volatile 修飾を取り除くキャストを行ってはならない。 【MISRA C:2012 R11.8】
4.4 単項演算	R2.5.2	単項演算子 "-" は符号なしの式に使用しない。
	R3.6.3	sizeof 演算子は、副作用がある式に用いてはならない。
	M1.5.1	関数識別子 (関数名) には、前に & を付けるか、括弧付きの仮引数リスト (空でも可) を指定して使用しなければならない。 【MISRA C:2004 16.9】
4.5 加減乗除	R3.2.1	除算や剰余算の右边式は、0 でないことを確認してから演算を行う。
4.6 シフト	R2.5.4	シフト演算子の右辺の項はゼロ以上、左辺の項のビット幅未満でなければならない。
4.7 比較	R2.1.1	浮動小数点式は、等価または非等価の比較をしない。
	R2.2.1	真偽を求める式の中で、真として定義した値と比較しない。
	M1.5.2	条件判定の式では、0 との比較は明示的にする。

文法による分類	No.	ルール
4 式		
4.8 ビット演算	R2.5.3	unsigned char型、またはunsigned short型のデータをビット反転 (~)、もしくは左シフト (<<) する場合、結果の型に明示的にキャストする。
4.9 論理演算	M1.4.1	&& 演算や 演算の右式と左式は二項演算を含まない式か () で囲まれた式を記述する。ただし、&& 演算が連続して結合している場合や、 演算が連続して結合している場合は、&& 式や 式を () で囲む必要はない。
	M1.8.1	論理演算子 && または の右側のオペランドには、副作用があってはならない。 【MISRA C:2012 R13.5】
4.10 3項演算	R2.3.2	条件演算子 (? : 演算子) では、論理式は括弧で囲み、戻り値は2つとも同じ型にする。
4.11 代入	R2.4.1	演算の型と演算結果の代入先の型が異なる場合は、期待する演算精度の型へキャストしてから演算する。
	R2.5.1	情報損失を起こす可能性のあるデータ型への代入 (= 演算、関数呼出しの実引数渡し、関数復帰) や演算を行う場合は、問題がないことを確認し、問題がないことを明示するためにキャストを記述する。
	M3.3.3	(1) 真偽を求める式の中で代入演算子を使用しない。 (2) 真偽を求める式の中で代入演算子を使用しない。ただし慣習的に使う表現は除く。
4.12 コンマ	M3.2.1	(1) コンマ式は使用しない。 (2) コンマ式は for 文の初期化式や更新式以外では使用しない。
4.13 優先順位と副作用	R3.6.1	変数の値を変更する記述をした同じ式内で、その変数を参照、変更しない。
	R3.6.2	実引数並び、及び2項演算式に、副作用をもつ関数呼出し、volatile変数を、複数記述しない。
	M1.4.2	《演算の優先順位を明示するための括弧の付け方を規定する。》
4.14 その他 (式)	R2.3.1	符号なし整数定数式は、結果の型で表現できる範囲内で記述する。
5 文		
5.1 if 文	R3.5.1	if-else if 文は、最後に else 節を置く。 通常、else 条件が発生しないことが分かっている場合は、次のいずれかの記述とする。 《(i) else 節には、例外発生時の処理を記述する。 (ii) else 節には、プロジェクトで規定したコメントを入れる。》

文法による分類	No.	ルール
5 文		
5.2 switch 文	R3.5.2	switch 文は、最後に default 節を置く。 通常、default 条件が発生しないことが分かっている場合は、次のいずれかの記述とする。 《(i) default 節には、例外発生時の処理を記述する。 (ii) default 節には、プロジェクトで規定したコメントを入れる。》
	M1.3.1	switch (式) の式には、真偽結果を求める式を記述しない。
	M1.3.2	switch 文の case ラベル及び default ラベルは、switch 文本体の複文（その中に入れ子になった複文は除く）にのみ記述する。
	M3.1.4	(1) switch 文の case 節、default 節は、必ず break 文で終了させる。 (2) switch 文の case 節、default 節を break 文で終了させない場合は、《プロジェクトでコメントを規定し》そのコメントを挿入する。
5.3 for・while 文	R2.1.2	浮動小数点型変数はループカウンタとして使用しない。
	R2.3.3	ループカウンタとループ継続条件の比較に使用する変数は、同じ型にする。
	R3.1.2	配列を順次にアクセスするループの継続条件には、配列の範囲内であるかの判定を入れる。
	R3.5.3	ループカウンタの比較に等価演算子 (==, !=) は使用しない。
	M1.9.2	《無限ループの書き方を規定する。》
	M3.1.1	繰返し文を終了させるために使用する break 文または goto 文は、1 つまでとする。 【MISRA C:2012 R15.4】
	M3.3.1	for 文の 3 つの式には、ループ制御に関するもののみを記述しなければならない。 【MISRA C:2004 13.5】
	M3.3.2	for ループの中で繰返しカウンタとして用いる数値変数は、ループの本体内で変更してはならない。 【MISRA C:2004 13.6】
5.4 その他 (文)	M2.1.2	if、else if、else、while、do、for、switch 文の本体はブロック化する。
	M3.1.2	(1) goto 文を使用しない。 (2) goto 文を使用する場合、飛び先は同じブロック、または goto 文を囲むブロック内で、かつ goto 文の後方に宣言されているラベルとする。
	M3.1.5	(1) 関数は、1 つの return 文で終了させる。 (2) 処理の途中で復帰する return 文は、異常復帰の場合のみとする。
	M3.2.2	1 つの文に、代入を複数記述しない。ただし、同じ値を複数の変数に代入する場合を除く。

文法による分類	No.	ルール
6 マクロ・プリプロセッサ		
6.1 #if系	M4.7.2	#ifdef、#ifndef、#ifに対応する#else、#elif、#endifは、同一ファイル内に記述し、《プロジェクトで規定したコメントを入れ対応関係を明確にする。》
	M4.7.3	#ifや#elifで、マクロ名が定義済みかを調べる場合は、defined（マクロ名）またはdefined マクロ名により定義済みかを調べる。
	M4.7.7	#ifまたは#elif前処理指令の制御式は、0または1に評価されなければならない。 【MISRA C:2012 R20.8】
6.2 #include	P1.4.1	#include指令の後には、<filename>または"filename"が続かなければならない。 【MISRA C:2012 R20.3】
	P1.4.2	《#includeのファイル指定で、<>形式と" "形式の使い分け方を規定する。》
	P1.4.3	#includeのファイル指定では、文字'、\、"、/*、//、及び: は使用しない。
	P1.5.1	#includeのファイル指定では、絶対パスは記述しない。
6.3 マクロ	M1.8.2	Cマクロは、波括弧で囲まれた初期化子、定数、括弧で囲まれた式、型修飾子、記憶域クラス指定子、do-while-zero構造にのみ展開されなければならない。 【MISRA C:2004 19.4】
	M4.7.1	演算子を含むマクロは、マクロ本体とマクロ引数を括弧で囲む。
	M4.7.5	マクロは、ブロック内で#define、または#undefしてはならない。 【MISRA C:2004 19.5】
	M4.7.6	#undefは使用してはならない。 【MISRA C:2012 R20.5】
6.4 その他（プリプロセッサ）	M1.8.3	#lineは、ツールによる自動生成以外では使用しない。
	M1.12.1	プリプロセッサが削除する部分でも正しい記述を行う。
	M5.1.2	(1) 前処理演算子#と##を使用してはならない。【MISRA C:2012 R20.10】 (2) #演算子の直後に続くマクロパラメータの直後に##演算子をつけない。 【MISRA C:2012 R20.11】
7 環境・他		
7.1 移植性	P1.1.1	(1) 言語標準の規格外の機能は使用しない。 (2) 言語標準の規格外の機能を使用する場合は、《使用する機能とその使い方を文書化する。》
	P1.1.2	《使用する処理系定義の動作はすべて文書化しなければならない。》 【MISRA C:2004 3.1】
	P1.1.3	他言語で書かれたプログラムを利用する場合、《そのインタフェースを文書化し、使用方法を規定する。》

文法による分類	No.	ルール
7 環境・他		
7.1 移植性	P1.2.1	プログラムの記述において、言語規格で規定している文字以外の文字を使用する場合、コンパイラの仕様を確認し《その使い方を規定する。》
	P1.2.2	言語規格で定義されているエスケープシーケンスだけを使用する。
	P2.1.1	C言語からアセンブリ言語のプログラムを呼び出す場合、インラインアセンブリ言語のみが含まれるC言語の関数として表現する、またはマクロで記述するなど、《局所化する方法を規定する。》
	P2.1.2	処理系が拡張しているキーワードは、《マクロを規定して》局所化して使用する。
7.2 性能	R1.3.4	restrict型修飾子は使用しない。 【MISRA C:2012 R8.14】
	E1.1.1	マクロ関数は、速度性能に関わる部分に閉じて使用する。
	E1.1.2	繰り返し処理内で、変化のない処理を行わない。
	E1.1.3	関数の引数として構造体ではなく構造体ポインタを使用する。
	E1.1.4	《switch文とするかif文とするかは、可読性と効率性を考えて選択方針を決定し、規定する。》
7.3 デバッグ用記述	M5.1.1	《デバッグオプション設定時のコーディング方法と、リリースモジュールにログを残すためのコーディング方法を規定する。》
7.4 その他	M5.2.1	(1) 動的メモリは使用しない。 (2) 動的メモリを使用する場合は、《使用するメモリ量の上限、メモリ不足の場合の処理、及びデバッグ方法などを規定する。》

付録 C 処理系定義の動作について

C 言語には、言語規格で未規定（不定）や未定義とされる記述があります（コラム参照）。

未規定とされるものの一部には処理系（コンパイラ）が動作を定義することになっているものがあり、これを「処理系定義の動作」といいます。処理系定義の動作となる項目は処理系ごとに動作が規定されています。すなわち同じ処理系であれば、いつも同じように動作します。

一方で、処理系が異なる場合には、ソースプログラム上の同じ記述が同じ動作とはならないことがあります。このため、プログラムの移植をしたり処理系を変更したりする場合に注意が必要です。また、特定の処理系だけに慣れている場合、そこでの処理系定義の動作を C 言語規格で規定された動作と思い込み、別な処理系を使う際に思わぬミスを引き起こすことがあります。したがって、開発を始める前に処理系定義の動作について確認しておくことが望ましいです。

処理系定義の動作（以下、処理系定義と略します）は、通常コンパイラのマニュアルに記述されています。ここでは、その代表的なものについて解説します。



処理系定義の代表例 1：動作環境

処理系定義の記述には、フリースタンディング環境（Freestanding environment）という言葉が登場することがあります。フリースタンディング環境とは平たく言えば OS のない環境です。このような環境では、プログラム起動時に呼び出す関数の名前と種類は処理系定義です。通常は main 関数が呼び出されますが、起動から main 関数に至るまでにどのような処理（見えない関数）が呼び出されているかは処理系によって異なります。

また、main 関数が終了したり exit などプログラムを中断したりした場合、その後どのような処理になるかも処理系定義です。まず、そのようなことを起こさないプログラムとすることが第一ですが、もしそうなった場合にどのような動作になるかは知っておく必要があります。



処理系定義の代表例 2：文字コード

文字コードとは、文字や記号をコンピュータで扱うために文字や記号一つ一つに割り当てられた固有の数値のことです。文字集合と対応する文字コードの集合の対応関係を文字コード体系といいます。どのような文字コード体系を利用できるかは処理系定義です。表 1 は ASCII とい

う7ビットの文字コード体系の表です。横軸が上位3ビット、縦軸が下位4ビットを表しています。たとえば英字Aは、上位3ビットが4で下位4ビットが1ですから、対応する文字コードは0x41と分かります。

ASCIIとは異なる文字コード体系として8ビットのEBCDIC（エビシディック）があります。表2はEBCDIC文字コード体系の表です。英字Aの文字コードはASCIIでは0x41でしたが、EBCDICでは0xC1です。

表1 ASCIIコード表

		上位ビット							
下位ビット		0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	.	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(8	H	X	h	x
	9	HT	EM)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

表2 EBCDICコード表

		上位ビット															
下位ビット		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	NUL	DLE	DS		SP	&	-						{	}	\	0
	1	SOH	DC1	SOS				/		a	j	~		A	J		1
	2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
	3	ETX	TM							c	l	t		C	L	T	3
	4	PF	RES	BYP	PN					d	m	u		D	M	U	4
	5	HT	NL	LF	RS					e	n	v		E	N	V	5
	6	LC	BS	ETB	UC					f	o	w		F	O	W	6
	7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
	8		CAN							h	q	y		H	Q	Y	8
	9		EM							i	r	z		I	R	Z	9
	A	SMM	CC	SM		¢	!		:								
	B	VT	CU1	CU2	CU3	·	\$,	#								
	C	FF	IFS		DC4	<	*	%	@								
	D	CR	IGS	ENQ	NAK	()	_	'								
	E	SO	IRS	ACK		+	;	>	=								
	F	SI	IUS	BEL	SUB		?	"									

このように、文字を表現するコードは使用する文字コード体系によって異なります。開発環境（ソースプログラムを扱う環境）と実行環境（実行ファイルが動作する環境）で使う文字コード体系が異なる場合、処理系がどのように対応しているか注意が必要です。

日本語（ひらがな、漢字など）には、1文字に対して2バイト以上からなる文字コードが割り当てられています。この文字コードにも複数の体系があり、使用する体系によって値が異なりますから、処理系がどのように対応しているか注意が必要です。現在多くのパソコンなどでは、日本語を表すために1文字を2バイトで表現するShift_JISという文字コード体系を使っています。

また、日本語も含めた多言語の文字を統一的に扱うためにUnicode（ユニコード）という文字コード体系が制定され、近年広く使われるようになっていきます。Unicodeでは、1文字を1バイトから6バイトの多バイトで表現し、その表現の仕方（符号化方式）によって次の三つの主要なコード体系があります。

1. UTF-8 : ASCIIと同じ部分は1バイトで、それ以外を2から6バイトの可変長で表現する
2. UTF-16 : 16ビットを単位として1文字を1単位 (16ビット) または2単位 (32ビット) で表現する
3. UTF-32 : 32ビットの単位一つだけの固定長で1文字を表現する

C言語では、多バイトの文字コード体系で表現される文字を、1文字を一定のビット数の整数値として扱えるように「ワイド文字」という文法を導入して機能拡張してきています。たとえばC99では、ワイド文字用の型 `wchar_t` を導入しており、それを扱うライブラリも追加されています。

ライブラリの例：

```
int vwprintf (const wchar_t * restrict format, va_list arg);
// ワイド文字に対応したprintf
```

またC11では、`char16_t`(2バイト長)、`char32_t`(4バイト長)という型を追加しています。ここで、`wchar_t`のサイズや、それぞれのワイド文字用の型がどのような文字コードに対応するかは処理系定義です。なお、C11では `__STDC_UTF_16__` と `__STDC_UTF_32__` というマクロを導入しており、このマクロが定義されている場合、`char16_t`と`char32_t`の符号化がそれぞれUTF-16とUTF-32であることを示しています。

使用する文字コード体系の違い以外でも、たとえばShift_JISで表現される漢字を使う場合の注意点があります。Shift_JISでは、2バイトで1文字を表現しますが、その2バイト目の値がASCIIの「\」(バックスラッシュ (または「¥」))と同じになるものがあります。たとえば「表」という文字のコードはShift_JISでは0x955cです。この先頭から2バイト目の0x5cはASCIIでは「\」に相当します(表1)。

Shift_JISをサポートしていないコンパイラの場合、2バイトで構成される1文字を単なる1バイトの文字の列とみなして「\」によるエスケープシーケンスの処理を行い、意図した表示にならないなどという不具合につながります。

例：

```
ソースコード： printf("表\n"); // バイト列：0x95 0x5c 0x5c 0x6e
//          → 0x5c 0x5c(\\) は 0x5c(\\) になる
出力          ： 表n          // バイト列：0x95 0x5c 0x6e
//          本来は「表」の後で改行することを意図
```



処理系定義の代表例 3：ポインタとアドレス

組込みソフトでは、アドレスの絶対値を扱うことが少なくありません。特定のアドレスにアクセスする場合にポインタを使いますが、この場合以下の例のようにポインタに整数を代入する（またはその逆の）演算が必要です。

```
unsigned char *addrp = (unsigned char *)0xffff0123L;
```

このような整数とポインタの変換が実際にどのような実行コードに変換されるかは、処理系定義です。また、アドレスの値をどのようなサイズで扱うかも処理系定義です。これらは処理系のみならず、実際にプログラムを実行するプロセッサのアーキテクチャにも大きく依存します。



処理系定義の代表例 4：配列

同じ配列の要素への二つのポインタを減算した結果のサイズは、必ずしもアドレスに対して適用されるビット幅のサイズとして保証されるわけではありません。処理系定義です。C99の言語規格 X3010:2003 (ISO/IEC9899:1999) では、ポインタ同士の減算結果のサイズの型として、`<stddef.h>` で `ptrdiff_t` という型を定義しています。



処理系定義の代表例 5：整数

符号付きの整数型を符号と絶対値による表示、2の補数表示、1の補数表示、のどれで表現するかは処理系定義です。したがって、たとえば、

```
if (( intVal & 0x80000000 ) == 0x80000000 ) { // 最上位ビットが1なら・・・
```

といった処理は、符号付き整数の最上位ビットが符号ビット（負値の場合に '1'）であるという表現を処理系が使っている場合にしか期待通りに動作しません。

また、規格外の値をトラップ表現、または通常値での表現のどちらにするかも処理系定義です。規格外の値とは、演算結果が変数のサイズに収まらなくなる場合の値を示します。符号なしの変数の場合は、演算結果が変数の表現範囲を超えると、その変数で表現できる最大値+1による剰余となります。たとえば符号なし8ビットの変数で、演算結果が257になった場合、8ビットの変数で表現できる最大値255に1を加えた256による剰余である1が演算結果となります。このような動作をラップアラウンドと言います。

一方、符号付きの変数では、演算結果が変数で表現できる範囲を超えた場合にオーバーフロー

となります。このとき、演算結果を符号なし変数の場合と同様に（たまたま変数に残った値として）表現する場合と、トラップ表現という特別な値で表現する場合があります。トラップ表現とは、システムが内部処理用に特別に定義している値のことで、2の補数表現を使っている場合は最上位ビットが1でそれ以外が0である値を使います。



処理系定義の代表例6：ビットフィールド

組込み用Cコンパイラでは、符号なし8ビットのサイズでビットフィールドを使用可能としたものがあり、マイコンの機能をビットに割り振っている内蔵レジスタのアクセスによく用います。

しかし、これは処理系定義であり、特定の処理系で正常に動作したものが別の処理系でも同じように動作する保障はありません。また、ビットの並びが上位ビットからか下位ビットからかも処理系定義です。

さらに、ビットフィールドを使ったからといって実際の実行コードが操作の対象（たとえば内蔵レジスタ）に対してビットアクセスをする命令になるかどうかは処理系定義です。1ビットのビットフィールドを使っている場合、実行コードでは、そのビットを含むバイトをアクセスしたリード・モディファイ・ライトになることもあり、思わぬ不具合の原因となりえます。



処理系定義の代表例7：volatile修飾型のオブジェクトへのアクセス

volatile修飾はコンパイラの最適化を抑制する場合に使用します。たとえば、割り込み待ちをする場合、割り込みハンドラで1になる変数をポーリングする方法があります。

```
while( InterruptFlag == 0 ) { ; }
```

この場合、変数InterruptFlagを1にする処理は、このループの中にありませんから、コンパイラは最適化を行って単なる無限ループに置き換えてしまうかもしれません。volatile修飾子は、このような最適化が実施されないよう抑制します。

volatile修飾したオブジェクトは処理系に未知の方法で変更されることを暗示しています。volatile修飾したオブジェクトへのアクセスを実行コードでどのように構成するかは、処理系定義です。



処理系定義の代表例 8：前処理指令

前処理指令に関しては、以下のような処理系定義項目があります。

- ・ `<>` または `""` で指定したヘッダー名の連なりを、ヘッダーまたは外部ソースファイル名に対応させる方法
- ・ 条件付きのインクルードを制御する定数式の文字定数の値が、実行文字セット中の同一の文字定数の値に一致するかどうか
- ・ 条件付きのインクルードを制御する定数式の単一文字の文字定数が、負の値をとることがあるかどうか
- ・ `#include` 指令内の前処理トークン（マクロ展開で生成されることもある）からヘッダー名を形成する方法
- ・ `#include` 処理の入れ子制限
- ・ 文字定数または文字列定数に `#` 演算子があるとき、汎用文字名で始まる `\` 文字の前に `\` 文字を挿入するかどうか
- ・ 非STDCの `#pragma` 動作
C99 (X3010:2003 (ISO/IEC9899:1999)) から、C 言語が標準で持っている `#pragma` 指令が追加されました。これをSTDC (標準C) の `#pragma` 指令といいます。
これら以外の `#pragma` についての動作は処理系定義です。
- ・ 翻訳の日付と時刻がわからないときの `__DATE__` と `__TIME__` の定義



処理系定義の代表例 9：その他

インライン指示やレジスタ修飾子を指定しても、それらが実際に有効になるかどうかは処理系定義です。

以上、いくつかの処理系依存項目について説明しましたが、これら以外については実際に開発に使用する（バージョンの）コンパイラのマニュアルを参照してください。

コラム：未規定の動作と未定義の動作

C 言語には、注意すべき動作に次の四つがあります。

1. 未規定の動作
2. 未定義の動作
3. 処理系定義の動作
4. 文化圏固有の動作

(詳細はC99言語規格「ISO/IEC 9899:1999 Programming Language C」Annex Jを参照)

未規定の動作と未定義の動作は似ていますが、それぞれ以下のように異なった意味の言葉です。

■未規定の動作

未規定の動作とは、文法的には正しいものの、その実行結果が処理の仕方によって複数あり得るものを言います。たとえば、関数への実引数への評価順序がこれにあたります。

```
printf("%d %d \n", i, i++ );
```

というコードの場合、`i`と`i++` のどちらが先に評価されるかによって表示結果が異なります。

未規定の動作にどのようなものがあるかについてはX3010:2003 (ISO/IEC9899:1999) のJ.1で列挙されています。未規定の動作となる記述はできるかぎり避けましょう。

■未定義の動作

未定義の動作とは、C 言語の規格上定義されていないもののことです。

たとえば、0で除算した場合の動作がこれにあたります。未定義の動作にどのようなものがあるかについてはX3010:2003 (ISO/IEC9899:1999) のJ.2で列挙されています。

言語の規格として定義されていませんので、決して未定義の動作となる記述をしてはなりません。使用する静的解析ツールにおいて、未定義の動作に関してどれが検出可能でどれが検出不可能かを把握しておく必要があります。

引用・参考文献

- [1] JIS X 25010:2013 システム及びソフトウェア製品の品質要求及び評価 (SQuaRE) — システム及びソフトウェア品質モデル
備考 ISO/IEC 25010:2011, Systems and software engineering-Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality modelsがこの規格と一致している。
- [2] CERT® C セキュアコーディングスタンダード、Robert C. Seacord 著 / JPCERT コーディネーションセンター 久保 正樹、戸田 洋三 訳、ISBN 9784048677967、アスキー・メディアワークス、2009 年 9 月
- [3] JIS X 3010:2003 プログラム言語 C
ISO/IEC 9899:1999, Programming languages – C、及び ISO/IEC 9899/Cor1:2001 がこの規格と一致している。なお「JIS X 3010:1996 プログラム言語 C」(ISO/IEC 9899:1990, Programming languages C、及び ISO/IEC 9899:1990/Cor 1:1994, ISO/IEC 9899:1990/Cor 2:1996, ISO/IEC 9899:1990/Amd 1:1995, C Integrity)がこの規格と一致)は旧規格である。
- [4] JIS X 3014:2003 プログラム言語 C++
備考 ISO/IEC 14882:2003, Programming languages – C++がこの規格と一致している。
- [5] "MISRA Guidelines For The Use Of The C Language In Vehicle Based Software", The Motor Industry Software Reliability Association, ISBN 9780952415665, April 1998, www.misra.org.uk/
- [6] "MISRA-C:2004 Guidelines for the use of the C language in critical systems", The Motor Industry Software Reliability Association, ISBN 9780952415626, October 2004, www.misra.org.uk/
備考 「自動車用 C 言語利用のガイドライン (第 2 版)」TP-01002、社団法人自動車技術会、2006 年 3 月が翻訳書である。
- [7] "MISRA C:2012 Guidelines for the use of the C language in critical systems", The Motor Industry Software Reliability Association, ISBN 9781906400101, The Motor Industry Research Association, March 2013, www.misra.org.uk
- [8] "Indian Hill C Style and Coding Standards", <ftp://ftp.cs.utoronto.ca/doc/programming/ihstyle.ps>
- [9] "comp.lang.c Frequently Asked Questions", <http://www.eskimo.com/~scs/C-faq/top.html>
- [10] 「組み込み開発者における MISRA-C 組み込みプログラミングの高信頼化ガイド」、MISRA-C 研究会編、ISBN 9784542503342、日本規格協会、2004 年 5 月
- [11] "GNU coding standards", Free Software Foundation, <http://www.gnu.org/prep/standards/>
- [12] "The C Programming Language, Second Edition", Brian W. Kernighan and Dennis Ritchie, ISBN 0-13-110362-8, Prentice Hall PTR, March 1988
備考 「プログラミング言語 C 第 2 版(訳書訂正版)」, B.W.Kernighan, D.M.Ritchie 著 / 石田晴久訳、ISBN 9784320026926、共立出版、1994 年 3 月が翻訳書である。
- [13] "Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs", Steve Maguire, ISBN 9781556155514, Microsoft Press, January 1993
備考 「ライティング ソリッドコード」、Steve Maguire 著 / 関本健太郎訳、ISBN 9784756103642、アスキー、1995 年 3 月が翻訳書である。
- [14] "The Practice of Programming", Brian W. Kernighan and Rob Pike, ISBN 9780201615869, Addison Wesley Professional, February 1999
備考 「プログラミング作法」、Brian W. Kernighan, Rob Pike 著 / 福崎俊博訳、ISBN 9784756136497、アスキー、2000 年 11 月が翻訳書である。
- [15] "Linux kernel coding style", <http://www.linux.or.jp/JF/JFdocs/kernel-docs-2.6/CodingStyle.html>
- [16] "C Style: Standards and Guidelines : Defining Programming Standards for Professional C Programmers", David Straker, ISBN 9780131168983, Prentice Hall, January 1992
備考 「C スタイル-標準とガイドライン」、David Straker 著 / 奥田正人訳、ISBN 9784303724807、海文堂出版、1993 年 2 月が翻訳書である。
- [17] "C Programming FAQs : Frequently Asked Questions", ISBN 9780201845198, Steve Summit
備考 「C プログラミング FAQ C プログラミングのよく尋ねられる質問」、Steve Summit 著 / 北野 欽一 訳、ISBN 9784775302507、新紀元社、2004 年 1 月が翻訳書である。
- [18] "C STYLE GUIDE (SOFTWARE ENGINEERING LABORATORY SERIES SEL-94-003)", NASA, Aug 1994, <http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf>

Ver. 1.0 執筆者

青木奈央	IPA/SEC（キャッツ株式会社）
上田直子	富士通株式会社
宇野 結	松下電器産業株式会社
大島健嗣	株式会社リコー
大野克巳	IPA/SEC（トヨタテクニカルディベロップメント株式会社）
宍戸文男	イーソル株式会社
八谷祥一	株式会社ガイア・システム・ソリューション
林田聖司	株式会社東芝
平山雅之	IPA/SEC 組込みエンジニアリング領域幹事（株式会社東芝）
二上貴夫	株式会社東陽テクニカ
古山寿樹	松下電器産業株式会社
三橋二彩子	NECエレクトロニクス株式会社
室 修治	IPA/SEC（横河デジタルコンピュータ株式会社）

Ver. 1.1 改版協力者

遠藤亜里沙	IPA/SEC（トヨタテクニカルディベロップメント株式会社）
遠藤竜司	三菱スペース・ソフトウェア株式会社
波木理恵子	株式会社オージス総研

Ver. 2.0 編著者・協力者

伊藤雅子	富士通株式会社
宿口雅弘	イーソル株式会社
舘 伸幸	名古屋大学
十山圭介	IPA/SEC
西山博泰	株式会社日立製作所
二上貴夫	株式会社東陽テクニカ
三橋二彩子	日本電気株式会社
三原幸博	IPA/SEC（アルパイン株式会社）

（50音順）所属は発行時のもの

監修

組込みソフトウェア開発力強化推進委員会

編集・著作

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター

SEC BOOKS

【改訂版】組込みソフトウェア開発向け

コーディング作法ガイド [C言語版] ESCR Ver. 2.0

2014 年 3 月 7 日 1 版 1 刷発行

2016 年 2 月 26 日 2 版 1 刷発行

編 者 独立行政法人情報処理推進機構
技術本部 ソフトウェア高信頼化センター
発行人 松本 隆明
発行所 独立行政法人情報処理推進機構
〒 113-6591
東京都文京区本駒込二丁目 28 番 8 号
文京グリーンコート センターオフィス
TEL : 03-5978-7543 FAX : 03-5978-7517
URL : <http://www.ipa.go.jp/sec>

©独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター 2016

ISBN978-4-905318-23-1

Printed in Japan

ISBN978-4-905318-23-1

C3055 ¥1619E

定価：本体1,619円＋税



9784905318231



1923055016194

IPA 独立行政法人 情報処理推進機構
技術本部 ソフトウェア高信頼化センター

SEC-TN13-001



古紙/バリア配合率70%以上を主成分としています。



この印刷物は、印刷用の紙へ
リサイクルできます。