MISRA C 2012 规范

前言

本文档翻译 MISRA C 2012 版的规则，以及作为代码编辑者对规则的理解和建议。本文档可作为MISRA C 2012 原始文档(英文文档)的补充，辅助理解 MISRA C 规则。译者补充的描述，建议内容会以红色标注，辅助理解的补充内容则以蓝色标注。

名词解释

Guideline

MISRA C 规范第 7 章、第 8 章中条目的统称，本想翻译为“规则”，但在第 6 章中对其做了分类，分为了“指令”和“规则”，故不可行。其直译可以为“指导方针”、“指南”、“准则”等，这份译档中， 取“准则”作为译名，不一定准确。

声明(declare)和定义(define)

声明一个变量只是将变量名标识符的有关信息告诉编译器，使编译器“认识”该标识符，但声明不一定引起内存的分配。而定义变量意味着给变量分配内存空间，用于存放对应类型的数据，变量名就是对相应的内存单元的命名。在 C/C++程序中，大多数情况下变量声明也就是变量定义，声明变量的同时也就完成了变量的定义，只有声明外部变量时例外。函数类似，声明只是告诉编译器有这个名称、类型的函数，而定义则是函数的真实实现。

连接/链接(linkage)

分为三类，外部连接(链接)(external linkage)、内部连接(链接)(internal linkage)和无连接(链接)(no linkage)。具体描述可参阅《程序员的自我修养》一书，这里仅描述它们的特征。

外部连接(链接)(external linkage)：对于变量，即无“static”修饰的全局可访问的变量；对于函数，即无“static”修饰的全局可调用的函数。它们即使没有在头文件中用“extern”做外部声明，仍然被识别为外部连接(链接)(external linkage)。

内部连接(链接)(internal linkage)：即由“static”修饰的全局变量和函数，它们尽可在所在文件内访问和调用，无法被全局访问/调用。

无连接(链接)(no linkage)：即函数内部变量。所有函数都是有连接(链接,linkage)的。内部变量包含临时变量和静态变量两种，它们的共同特征是均无法在本函数外被访问。

在下面的描述中，将不直接描述连接/链接，而已以全局，局部为描述，以便于理解。

对象(object)

本规范的编制，具有普适性，故会出现如“对象”、“类”这些标准 C 中不提及的概念，对象在 C 语言中的直接对应是变量。当前对象不仅仅是变量，但本译文仅限考虑标准 C(准确的说是嵌入式 C)，故不过多描述，我们将其当成“变量”理解即可。

1 愿景(The vision)

MISRA C 指南定义了C 语言的子集，以删除或减少犯错的机会。此 C 语言子集被许多用于开发与安全相关的软件的标准要求或建议使用，并且它也可以用于开发具有高完整性或高可靠性要求的任何应用程序。

除了定义此子集外，这些 MISRA C 指南还提供了：

◇ 为正在开发C 计划的人准备的教育材料；

◇ 工具开发人员参考资料。

以前的 MISRA C 版本是基于 1990 年的 ISO C 定义的。由于现在 1999 年的 ISO 定义已在不同程度上被嵌入式实现所采用，人们认为发布 MISRA C 的新版本的时机已经成熟。 它认可了 1999 年的 ISO 定义。

上一版中提供的指南的各个方面均已进行了全面审查，并在适当时进行了改进。 此第三版还结合了根据早期版本的指南用户提供的反馈而创建的材料。

第三版的主要变化是将第二版的基础类型概念发展为基本类型。 使用新的基本类型概念，有可能开发出一套准则来为 C 语言带来更强的键入性。

第三版 MISRA C 的愿景是：

◇ 采用 1999 年的C 语言 ISO 定义，同时保留对 1990 年的较早定义的支持；

◇ 更正第二版中的任何已知问题；

◇ 添加有充分理由的新准则；

◇ 改进现有准则的规范和理由；

◇ 删除任何理由不足的准则；

◇ 增加静态分析工具可以处理的准则数量；

◇ 提供有关准则对自动生成代码的适用性的指南。

2 MISRA C 的背景(Background to MISRA C)

2.1 C 语言的普及

C 编程语言之所以受欢迎是因为：

◇ C 编译器可用于许多处理器;

◇ C 程序可以编译为高效的机器代码;

◇ 它由国际标准定义;

◇ 它提供了直接或通过语言扩展来访问目标处理器的输入/输出功能的机制;

◇ 在关键系统中使用 C 有相当丰富的经验;

◇ 静态分析和测试工具广泛支持它。

2.2 C 语言的缺陷

尽管非常流行，但 C 语言具有一些缺点，以下各节将对此进行讨论。

2.2.1 语言的定义

ISO C 语言标准并未完全指定语言，而是将某些方面置于实现的控制之下。这是有意为之，部分原因是希望兼容广泛不同的目标处理器的许多现有实现。

因此，在 C 语言的某些部分：

◇ 行为是未定义的;

◇ 行为是未指定的;

◇ 一个实现可以自由选择自己的行为，前提是它被记录在案。

依赖未定义或未指定行为的程序不一定能保证以可预测的方式运行。

过度依赖于实现定义的行为的程序可能很难移植到其他目标。如果无法配置分析器进行处理，则实现定义的行为的存在也可能会妨碍静态分析。

2.2.2 语言的滥用

C 程序可以结构化和易于理解的方式进行布局，同时，C 也让程序员编写出难以理解的晦涩代码变得非

常容易。

C 的操作规范(的过分灵活)使得程序错误很难被编译器检测到。例如，以下两个代码片段都是合法的，因此编译器不可能分辨出是否已经错误的使用了其中一个代替了另一个：

```
if (a == b) /* 检测 a 和 b 是否相等                    */
if (a = b) /* 将 b 的值赋给 a，然后检测 a 是否非 0 */
```

2.2.3 语言的误会(代码的误读)

C 语言的某些部分经常被程序员误解。例如，C 语言比其他某些语言拥有更多的运算符，因此具有大量不同的运算符优先级，而其中有些并不直观。

C 提供的类型规则也会使熟悉强类型语言的程序员感到困惑。例如，操作数可以被“提升”为更宽的类型，这意味着从操作中得出的类型不必与操作数相同。

2.2.4 运行时错误检查

C 程序可以被编译为小型高效的机器代码，但其代价是，运行时检查的程度非常有限。 C 程序通常不对常见问题提供运行时检查，例如算术异常(例如零除)，溢出，指针有效性或数组绑定错误。C 的哲学是程序员负责显式地进行此类检查。

3 工具选择(Tool selection)

3.1 C 语言及其编译器

在开始编写本文档时，C 语言的 ISO 标准是 ISO / IEC 9899：1999 [8](已由[9]，[10]和[11]进行了更正)，以下将该语言的版本称为 C99。但是，仍有许多编译器仍然支持其前身(由 ISO / IEC 9899：1990 [2]定义并由[4]，[5]和[6]修订并更正，以下称为 C90)。另外，还有一些编译器则提供了在两种语言版本之间进行选择的选项。

当前针对 C 语言的 ISO 标准是 ISO / IEC 9899：2011 [13](已通过[14]进行了更正)。但当该标准发布时，有关 MISRA C 指南的工作已接近完成，因此无法合并此更高版本的指南。

小贴士：使用 MISRA C 不需要阅读相关标准的副本，但(阅读相关标准的副本)会有帮助。

C90 和C99 之间的选择可能受在项目上重用的遗留代码数量以及目标处理器的编译器是否支持等因素影响。

为项目选择的编译器应该满足 C 语言所选版本的符合规范的独立实现的要求。该编译器可能会提供超出要求的功能(例如：提供语言的所有功能(独立实现仅需提供明确定义的子集))，也可能提供语言标准所允许的扩展。

理想情况下，编译器开发人员应提供对编译器确实符合要求的确认，例如，通过提供运行的符合性测试和获得的结果的详细信息。

有时候编译器的选择可能有限，其质量可能未知。如果证实难以从编译器开发人员那里获取信息，则可以采取以下步骤来协助选择过程：

◇ 检查编译器开发人员是否遵循适当的软件开发流程，例如一项符合 ISO 9001：2008 [21]的要求，该要求已通过ISO 90003：2004 [22]进行了评估；

◇ 阅读对编译器和用户体验报告的评论；

◇ 审查用户规模和使用编译器开发的应用程序类型；

◇ 执行和记录独立的验证测试，例如通过使用一致性测试套件或编译现有应用程序。

小贴士：某些流程标准在某些情况下要求对编译器进行鉴定，例如 IEC 61508：2010 [32]，ISO 26262：2011 [23]和 DO-178C [24]。

3.2 分析工具

通过人眼检查源代码是否符合 MISRA C 是可能的，但这么做可能会非常耗时且非常容易出错。所以，任何针对 MISRA C 检查代码的实际过程都需要使用至少一种静态分析工具，推荐使用国产工具-Pinpoint。

尽管分析工具的验证与编译器的验证稍有不同，但适用于编译器选择的所有因素也适用于静态分析工具的选择。理想的分析工具应该能够：

◇ 发现所有违反 MISRA C 准则的行为；

◇ 不产生“误报”，即只会报告真正的违规行为，不会报告非违规或可能的违规行为。

由于第 6.5 节中所述的原因，永远不可能生产出符合这种理想行为的静态分析工具。因此，在选择分析工具时，最大化的检出违规行为，并最大程度的减少误报的能力，是一个重要因素。

可用的工具种类繁多，执行时间从几秒到几天不等。总的来说，花费时间较少的工具，比起需要消耗大量时间的更容易产生误报。在选择工具时，还应考虑分析时间与分析精度之间的平衡。

每个分析工具的侧重点各有不同。有些可能是通用的，有些则可能专注于对潜在问题的子集进行彻底的分析。因此，有可能需要使用多个工具来保证最大化问题的覆盖面。

小贴士：*某些过程标准，包括 IEC 61508:2010[32]、ISO 26262:2011[23]和 DO-178C[24]，在某些情况下要求对验证工具(例如静态分析仪)进行鉴定。*

4 必备知识(Prerequisite knowledge)

4.1 培训

为了确保产生C 源代码的人员具有适当水平的技能和能力，应提供以下方面的正式培训：

◇ 嵌入式应用程序中C 语言的使用

◇ 使用 C 语言编写高完整性和安全相关的系统

由于编译器和静态分析工具是复杂的软件，因此也应考虑提供使用它们的培训。通常情况下，选用的静态分析工具可能会提供专门针对 MISRA C 的使用培训。

4.2 了解编译器

在本文档中，术语“编译器”，被 ISO C 标准[2]、[8]称为“实现”，表示编译器本身及其相关工具，例如链接器，库管理器和可选的文件格式转换工具。

编译器可以提供控制其行为的各种选项。了解这些选项的影响非常重要，因为它们的使用或不使用可能会产生影响：

◇ 语言扩展的可用性

◇ 编译器与 ISO C 标准的一致性

◇ 程序所需的资源，特别是处理时间和内存空间;

◇ 编译器中的缺陷被暴露的可能性，例如在执行复杂的高度优化代码转换时可能发生的情况。

理解编译器如何实现 ISO C 标准中称为“实现定义”的 C 语言的那些特性非常重要。同样的，理解编译器可能提供的任何语言扩展也很重要。

编译器开发人员可能会提供(并维护)已知会影响编译器的缺陷清单及其可用的解决方法。在使用该编译器启动项目之前，了解该列表的内容显然是有利的。如果无法获得这份清单，则应在发现缺陷或疑似缺陷的情况下，记录一份本地清单，并将该发现报告给编译器开发人员。

4.3 了解静态分析工具

为了解项目中使用的每个静态分析工具，我们应阅读它们的文档以了解：

◇ 如何配置分析工具以匹配编译器的实现定义的行为，例如:整数类型的大小；

◇ 分析工具能够检查的 MISRA C 准则（第 5.3 节）；

◇ 是否可以将分析工具配置为支持将要使用的语言扩展

◇ 是否可以调整分析工具的行为，以在分析时间和分析精度之间取得不同的平衡

工具开发人员可能会提供(并维护)已知会影响工具的缺陷清单及其可用的解决方法。在使用该工具开始项目之前，了解此列表的内容显然会很有利。如果无法获得这份清单，则应在发现缺陷或疑似缺陷的情况下，记录一份本地清单，并将该发现报告给工具开发人员。

5 采用和使用 MISRA C(Adopting and using MISRA C)

5.1 采用

MISRA C 应该从项目之初即采用。

如果一个项目是建立在经过验证且具有良好记录的现有代码的基础上的，那么使其符合 MISRA C 的工作所带来的好处可能会被其引入缺陷的风险所抵消。在这种情况下，应该分析可能获得的净收益，以判断是否采用 MISRA C。

5.2 软件开发过程

MISRA C 旨在在文件化软件开发过程的框架内使用。虽然 MISRA C 可以单独使用，但在流程中使用以确保其他活动已正确执行时具有更大的好处，例如：

◇ 软件要求(包括任何安全要求)是完整、明确无误的；

◇ 到达编码阶段的设计规范符合要求，且不包含其他功能；

◇ 编译器生成的对象模块的行为与相应设计中指定的行为一致；

◇ 对象模块已经过单独或一起测试，以识别和消除错误。

程序员应在将代码提交进行审查或单元测试之前使用 MISRA C。在生命周期后期检查 MISRA C 合规性的项目可能会花费大量时间重新编码、审查和测试。因此，在软件开发过程中，应尽可能早的使用 MISRA C 原则。

关于安全相关软件开发过程的详细讨论不在本文档的范围内。开发过程的示例可以在 IEC 61508:2010 b[32]、ISO 26262:2011[23]、DO-178C[24]、EN50128:2011[33]和 IEC 62304:2006[34]等标准和指南中找到。本节的其余部分将处理 MISRA C 和软件开发过程之间的交互。

5.2.1 MISRA C 要求的流程活动

为使用 MISRA C，有必要指定并记录以下内容：

◇ 合规矩阵(compliance matrix)，标识如何检查每条 MISRA C 准则的遵从性；

◇ 背离过程(deviation process)，通过该过程可以授权和记录合理的不合规情况。

软件开发过程还应该记录为避免运行时错误而采取的步骤，并证明已经避免了这些错误。例如，它应该包含下面这些演示过程的描述并将其记录下来：

执行环境为程序提供了足够的资源，特别是处理时间和堆栈空间；

程序区域中没有运行时错误，例如：通过检查计算过程的输入范围的代码，保证了没有计算溢出。

5.2.2 MISRA C 预期的流程活动

人们认识到，一致的风格有助于程序员理解其他人编写的代码。但是，由于风格是单个组织的问题，MISRA C 并没有提出任何纯粹与编程风格相关的建议。开发和使用本地样式指南作为软件开发过程的一部分是被期望的。

许多软件过程标准建议使用度量标准，以作为识别可能需要额外检查和测试工作的代码的手段。但是，收集的度量的性质及其相应的阈值将由行业，组织和/或项目的性质确定。因此，本文档不提供关于软件度量的任何指导。

5.3 合规矩阵

为了确保代码符合 MISRA C 的所有准则，合规矩阵应运而生。该矩阵列出每一条准则，并明确其检测方案。对大多数准则而言，最简单、最可靠也最具成本效益的检查方法是使用静态分析工具、编译器，或者两者结合使用。如果准则不能被完全覆盖，则需要手动检查。

一些自动代码生成工具的开发人员会随工具一起提供其合规性声明。该声明确定了如果工具按照开发人员的指定进行配置，则生成的代码不会违反的 MISRA C 准则。例如，它可能：

※列出遵从的准则；

※声明不违反所有强制性和要求的准则。

使用自动代码生成工具的的合规声明，可以显著减少其他方式需要检查的 MISRA C 准则的数量。

合规矩阵的示例见表 1，准则的摘要见附录 a，这些准则可用于帮助生成完整的合规矩阵。

Guideline	Compilers		Checking tools		Manual review
	‘A’	‘B’	‘A’	‘B’	Procedure x
Dir 1.1					
Dir 2.1	No errors	No errors			
…					

Rule 4.1			Message 38		
Rule 4.2				Warning 97	
Rule 5.1	Warning 347				
…					

表 1：合规矩阵示例

合规矩阵就位后，就可以配置编译器和静态分析工具，并其生成需要检查的消息列表。

检查过程中，每一款工具都应记录一下信息：

- 版本号；
- 工具调用使用的选项；
- 该工具使用的任何配置数据。

5.3.1 编译器配置

如果编译器提供 C90、C99 间的选择，则我们必须为指定的项目对其进行配置。类似的，如果编译器提供了目标选择，则我们必须为项目具体需要对该选择项进行配置。

我们应该仔细审查和选择编译器的优化选项，以确保获得执行速度和代码尺寸间的平衡。注意，使用更激进的优化选项，可能会增加暴露编译器缺陷的风险。暴露编译器缺陷可能会使我们为之付出巨大代价， 比如某些计算无法得出正确结果，而一般这种缺陷我们很难在最初就发现，当发现时，可能代价已经产生了，更有甚者，后果发生了，我们也不一定能查得出问题出在哪。

即使编译器未被用于确保合规性，它也可能在编译过程中产生表示存在潜在缺陷的消息。如果编译器能够控制它产生的消息的数量和性质，则我们应该检查这些消息，并配置适当的级别(即配置哪些消息可产生，哪些消息可忽略，一般编译器不会给我们提供非常详细的选择选项，而是提供一个级别供选择)。

5.3.2 静态分析工具配置

编译器通常针对特定的一款或一个系列的处理器，而静态分析工具往往是通用的。因此，配置每个静态分析工具以适配编译器就变得很重要了。例如，静态分析工具需要知道整数类型的大小(整数“int”类型的大小，因处理器、编译器的不同而不同)。

如果静态分析工具同时支持 C90 和 C99，则我们需要配置其匹配我们的具体项目。

若可能，静态分析工具应支持语言扩展。若其无法支持，则我们应该采用替代工具来检查代码是否符

合扩展语言。扩展语言：编译工具支持的，非标准 C 语言规定的代码格式，如：在当初 C99 规范未形成时的“//”注释，51 单片机的“bit”类型等。

静态分析工具还需要被配置以使其能够正确检查某些准则。例如，除非项目使用了“_Bool”来定义布尔类型，否则需要手动配置静态分析工具，使其能够正确识别布尔类型(附录 D)，才能正确检查相关 MISRA C 准则。

5.3.3 检测消息

合规性检查过程或编译过程中生成的消息可按如下分类：

1. 确认违反某一条 MISRA C 准则；
2. 可能违反某一条 MISRA C 准则；
3. 错误诊断出的某一条MISRA C 违规消息；
4. 非MISRA C 违规，但也报出的消息*(比如编译时产生的warning)*。

此四类描述很拗口，可能翻译不准确，可阅读原文增强理解。

针对(1)类消息，首选的解决措施是更正源代码，以使其符合 MISRA C 规范。如果不希望或是不可能使其合规，那么此处可能需要提出偏差记录(见章节 5.4)。

(2)、(3)、(4)类消息，也应予以重视。最简单、最快捷的解决方案时修改源代码以消除这几类消息。但是，这么做不一定总是可行，在这种情况下，应保留调查记录。记录的目的是：

- 针对(2)类消息，解释为什么尽管可能存在违规行为，但代码符合 MISRA C 规范；
- 针对(3)类消息，解释并尽可能获得工具开发人员同意，即该类消息为误诊；
- 针对(4)类消息，证明为什么可以将其安全忽略。

此类检测的所有记录应由具有适当资格的技术机构审查和批准。

5.4 背离说明

某些情况下，实际代码可能有必要背离本文档给出的指导原则。例如，在固定地址访问内存映射 I/O 端口的常见方法，不符合 MISRA C，因为它将整数强制类型转换为了指针：

```
#define PORT (*(volatile unsigned char *)0x0002)
PORT = 0x10u;
```

如 NXP、STM32 的库文件，寄存器也是以这种方式实现的内存映射。

正确的授权和记录这种背离非常重要。我们可以通过正式的授权程序来防止个别程序员背离指导方针。

正式的记录并保存，是种很好的做法，它可以用于为软件任何一种安全性论证提供依据。

背离说明应在软件开发过程中正式确定*(形式化)*。因为使用的方法因组织而异，所以 MISRA C 没有强制规定任何规程。例如：

- 用于提出，审查和批准背离的物理方法；
- 在批准背离之前，所需的审查和证据程度可能会根据准则的不同而有所不同。

每一条 MISRA C 准则，都对应一个类别，而该类别包含了可能适用的背离方式(见章节 6.2)。背离分为两类：项目背离和特定背离。

项目背离：当 MISRA C 在特定的使用情况下发生背离时，我们可以使用项目背离。例如：当内存映射 I/O 被使用在通信驱动中时，那么可以适当的扩展项目背离适用范围，以使内存映射 I/O 的使用范围涵盖该通信驱动的文件。项目背离通常在项目早期提出。

特定背离：当单个文件中的单个实例背离 MISRA C 时，使用特定背离。

一份背离说明应包括：

- 被背离的 MISRA C 准则；
- 允许背离的情况：对于项目背离，它可能是一个通用的描述，如“模块X 中的所有代码”；而对于特定背离，需要标识出背离代码发生的具体位置；
- 背离的理由：包括但不限于与其他可能性相比，背离的风险程度的评估。
- 确保安全的证明：例如，提出风险不会发生的证据，以及其所需的额外审查和测试；
- 背离的潜在后果以及已采取的缓解措施。

在附录 I 中有一份实例背离说明。

5.5 合规声明/遵从性声明

我们不能为组织做合规声明，只能为项目做合规声明。

当声明项目符合 MISRA C 时，开发人员应声明存在证据表明：

1. 已完成合规矩阵，该矩阵需描述 MISRA C 条目执行手段；
2. 项目中所有的代码均符合 MISRA C 规范，或者被批准的偏差；
3. 每个被批准的偏差均有记录；
4. 本节以及第 3 节和第 4 节中给出的指导已被采纳(*这部分没弄明白*)；
5. 工作人员技术娴熟、经验丰富。

小贴士：*在声明该代码符合规范时，我们是假设工具或评审人员已识别出所有不符合项。 由于其他工*

具或评审者可能会识别出不同的违规情况，因此我们必须认识到合规声明不是绝对的，而是取决于所使用的检查流程。

第(4)项所述指引的摘要以附录 F 的核对表的形式提供。

准则简介(Introduction to the guidelines)

本章对第 7 章和第 8 章(本文档的主要章节)中的规范做了介绍。

6.1 准则分类

每个 MISRA C 准则都可以被归类为“规则”和“指令”。

指令是一种指导原则，无法提供执行检查合规性所需的完整描述。 为了能够执行检查，还需要其他信息，例如可能在设计文档或要求规范中提供的信息。 静态分析工具可能有助于检查是否符合指令，但不同的工具可能会对违规的内容进行广泛的不同解释。

规则是提供完整的要求描述的准则。 应该可以检查源代码是否符合规则而无需任何其他信息。 特别指出，静态分析工具应该能够检查是否符合第 6.5 节中描述的限制的规则。

第 7 章包含所有指令，第 8 章包含所有规则。

6.2 准则要求级别分类

6.1 节是 “classification”，6.2 节是“category”，两者其实都是分类的意思，这里只能根据小节里的内容强行区别了。

每一条 MISRA C 准则都有一个“强制”，“必要”或“建议”属性类别，其含义如下所述。 除了这种基本分类之外，该文件没有给出也不打算暗示每个准则的重要性等级。 所有定义为“必要”的准则，无论是规则还是指令，都应被视为同等重要，所有“强制”和“建议”准则也应如此。

6.2.1 强制准则

声明符合本文档的C 代码应符合每项强制准则，且不允许有偏离。

小贴士：如果检查工具产生诊断消息，这并不一定意味着由于第 6.5 节中给出的原因违反了准则。

6.2.2 必要准则

声明符合本文件的 C 代码应符合所有必要准则，如果有不符合项，则应如第 5.4 节所述，有正式的偏差声明。

组织或项目可以有选择的将必要准则当做强制准则来执行。

6.2.3 建议准则

这些准则是建议执行。 然而，“建议”的地位并不意味着可以忽略这些条目，而是应在合理可行的范围内遵循这些条目。 建议准则不需要正式偏差授权，但如果不遵循正式偏离程序，则应制定替代安排以记录违规情况。

组织或项目可以有选择的将建议准则当做强制或必要准则来执行。

6.3 准则的组织原则

文档中的准则是根据 C 语言的不同主题来组织的。但是，这不可避免的会存在重叠现象，即一个准则可能与多个主题相关。在这种情况下，准则会被放在最相关的主题下。

6.4 冗余准则

本文档中的部分准则，涉及到在其他章节里禁止或不建议使用的语言功能。这是有意为之。可能用户会选择使用该功能(方法是针对对其有要求的准则提出偏差并记录，或者选择不遵循“建议准则”)，在这种情况下，限制该功能使用的次一级准则就变得有意义了。

6.5 规则可判定性

每一个强制、必需和建议**规则**，都被分类为可判定或不可判定的。这种分类方法描述了静态分析工具在理论上回答“此代码是否符合此规则？”的能力。而**指令**没有以这种方式分类，是因为我们无法设计出一种仅凭源代码就能确定其合规性的算法。

如果程序在每种情况下都可以用“是”或“否”来回答其合规性，则该规则是可判定的，否则，即为不可判定。对于判定是否合规取决于运行时属性的，规则可能是不可判定的，例如：

- 对象的值；
- 控制是否能达到特定程序中的特定点。

对静态代码分析而言，可判定的规则非常有用。如果我们我们配置了正确且无缺陷的静态分析工具，则：

- 分析工具报告了违规，可确认为实际违规；
- 分析工具报告了合规，也可认定为确实合规。

下面是可判定规则的一些例子：

- 规则 5.2 ：标识符的名称和作用域；
- 规则 11.3：源指针和目标指针的类型；
- 规则 20.7：宏扩展结果的语法形式；

静态分析工具检测不可判定的规则的能力不尽相同：

- 一个不可判定规则的违规报告，不表明代码一定违规，有些分析工具采用的报告策略是：报告可能的违规行为来提醒用户此处存在违规风险；
- 未报告的不可判定规则的违规，也不表明代码中一定没有该违规。

不可判定规则的一些实例：

- 规则 12.2：移位运算符的右侧操作数的值；
- 规则 2.1：知道用不可达的某个点。

我们应按 5.3.3 节所描述的方案，制定一个过程来分析静态分析的结果并将其记录。针对可判定和不可判定规则有关的任何输出过程，我们都应特别注意。

6.6 分析范围(分析的作用域)

每个规则都需要根据要检查的代码量进行分类，以便检查违规。至于可判定性，分析范围的概念不适用于指令(*指令没有可判定性属性*)。

规则的分析范围可以分类为“单个编译单元”和“系统”。

若一个规则被归类为能在“单个编译单元”的基础上检查，那么可以通过独立的检查每个编译单元来检查项目中的所有违规行为。例如，在一个编译单元中存在不含“default”标签的 switch 语句(规则 16.4)， 它对其他编译单元是否包含此类的 switch 语句没有影响。

编译单元可以直观的理解为一个c 文件，c 语言是逐个c 文件进行编译，然后进行链接，最终生成目标文件的。h 文件是辅助文件，不被直接编译。

若某个规则被归类为需要基于“系统”来检查，则在编译单元中的违规情况需要检查的范围要多于涉及的编译单元。“系统”的规则最好通过分析所有源代码来检查其合规性，尽管可能仅检查源代码的某个子集也可能会发现违规。例如，若一个项目中有 A、B 两个编译单元，可以检查每个编译单元中对象的所有声明和定义是否使用相同的类型名和限定符(规则 8.3)。但这不能保证 A 中的声明和定义使用与 B 中相同的类型名和限定符。因此，需要检查待编译并链接到可执行文件的所有源代码，才能确保符合这条规则。

所有不可判定的规则都要基于“系统”来检查，因为在一般情况下，都需要其他编译单元的行为信息。例如，在下面的函数 g 中，是否在使用 x 之前设置了自动对象 x 的值(规则 9.1)，将取决于另一个编译单元中定

义的函数 f 的行为：

```
extern void f(uint16_t *p);
uint16_t y;
void g(void)
{
    uint16_t x; /* x 未赋值 */
    f ( &x );   /* f 可能会修改其参数指向的对象 */
    y = x;      /* x 是否被赋值不可知 */
}
```

6.7 多元组织的项目

一个项目可能涉及来自各种组织的代码，例如：

- 编译器开发者提供的标准库代码；
- 设备供应商提供的底层驱动代码；
- 操作系统和专业供应商提供的高级驱动程序代码；
- 协作组织间基于专业知识共享的应用程序代码。

标准库代码可能更关心执行效率。它可能依赖于实现细节和未指明的行为，例如：

- 将指向对象类型的指针转换为指向其他对象类型的指针；
- 堆栈帧的布局，特别是帧内功能参数的位置；这里明显指的是通信帧的位域定义
- 在 C 语言中嵌入汇编语言的语句。

由于它们已经是实现的一部分，且其功能和接口已经在 ISO C 标准中定义，因此不需要要求标准库代码符合 MISRA C。除非个别准则中有规定，否则标准库头文件、以及处理标准库头文件期间包含的任何文件都不需要遵守 MISRA C。但是，限制了标准库头声明和宏所提供的接口的准则，仍然适用。例如，基本类型规则适用于传递给标准库中指定的函数的参数类型及其结果。

所有的其他代码应最大程度的遵守 MISRA C 准则。如果所有源代码均可获取，则可以检查整个程序是否合规。但是，在许多项目中，开发组织不可能获得所有源代码。例如，合作组织可以共享功能规范、接口规范和目标代码，但有可能通过不共享源代码来保护自己的知识产权。当只有部分源代码可用于分析时， 应采用其他技术来确认合规性。例如：

- 提供商业代码的组织可能愿意发布其 MISRA C 合规声明；
- 合作组织间则可能：

- 商定适用于各自源代码的通用程序和工具；
- 相互提供源代码的存根版本，以允许进行跨组织检查。

一个项目应定义它将使用以下哪种方法来处理其他组织提供的代码：

1. 以与内部开发代码相同的方式 - 适用于提供源代码的情况；
2. 以与标准库相同的方式 - 适用于提供适当的 MISRA C 合规声明的组织；
3. 对头文件及其接口内容执行合规检查 - 假定没有合规声明，且除了头文件外无源代码可用；

在情况(3)中，应在使用代码前采用适当的附加验证和确认技术。

小贴士：某些流程标准可能包括管理多组织开发的更广泛的要求，如 ISO 26262[23] 第 8 部分第 5 条“分布式开发的接口”。

6.8 自动生成的代码

MISRA C 准则适用于自动生成的代码。自动代码生成工具的开发人员和使用工具生成代码模型的开发人员都应承担合规责任。由于可能存在多个建模程序包，且每个建模程序包都可能有多个代码生成工具，因此规范无法为 MISRA C 准则单独分配职责。针对建模软件包和代码生成工具，用户可采用如 MISRA AC GMG [17]，MISRA AC SLSF [18]和 MISRA AC TL [19]的相关准则。

这段需要额外查找更多资料，以确认自动生成代码的规范到底是怎么遵循。

应用于 MISRA C 准则的类别在应用于自动生成的代码时不必与应用于手动编写的代码相同。 在进行区分时，必须注意一些建模程序包允许将手动编写的代码注入到模型中。 在确定准则的适用类别时，不应将此类代码视为已自动生成。

附录 E 标识了适用于自动生成代码的类别与手动生成代码的类别不同的 MISRA C 准则。 它还说明了对自动代码生成器开发人员的文档要求。

6.9 准则格式介绍(介绍条目格式)

本文档的各个准则以下述格式表示：

标识 要求描述文字

[参考文献]

级别 要求级别

分析 可判定性，范围

适用 Cxx

其中：

- “标识”是准则的唯一标识；
- “要求描述文字”是准则本身的内容；
- “来源参考”表示在适用的情况下，制定该条目或条目组的依据参考。可参阅第 6.10 节，以获取这些参考文献的解释和原始资料的要点；
- “要求级别”是“强制”、“必要”或“建议”之一，见 6.2 节描述；
- “可判定性”是“可判定”或“不可判定”之一，见 6.5 节描述；
- “范围”是“系统”或“单个编译单位”之一，见 6.6 节描述；
- “Cxx”是“C90”或“C99”中的一个或多个，以逗号分隔，表示该准则适用于哪些 C 语言版本。

小贴士：指令没有可判定性和分析范围的分析属性，因此会省略“分析”行。

另外，每个条目或条目组都有说明文字，该文字对准则要解决的基本问题进行了一些解释，并举例说明了如何应用准则。

在辅助文本中，可能会有一个题为“**展开**”(Amplification)的标题，其后是该准则的更精确描述的文本。 **展开**是规范性的，如果与标题冲突，则**展开**优先。 这种机制很方便，因为它允许使用简短的标题传达复杂的概念。

在辅助文字中，可能会有一个标题为“例外”的标题，其后是描述该规则不适用的情况的文字。 例外的使用允许简化一些准则的描述。 重要的是要注意，例外情况是准则不适用的情况。 由于例外而符合准则的代码不需要(执行)偏差(规程)。

由于假定读者具有该语言的实用知识，因此，本文不打算将其用作相关语言功能的教程。有关语言功能的更多信息，请查阅语言标准的相关章节或其他 C 语言参考书，可能会为理解准则提供更多帮助。*这里省略了一段很拗口的内容，但意思已基本描述清楚了。*

在准则及其描述文字中，下述字体用于标识 C 关键字和 C 代码：

- 用等宽字体显示，且加底纹；*这里与原文不一致，描述的是本文档的格式。*

小贴士：在引用代码的地方，片段可能不完整(例如，没有其主体的 if 语句)。 这是为了简洁起见。

在代码片段中，假定使用以下 typedef 类型(这符合指令 4.6)：

```
uint8_t /* unsigned 8-bit integer  8bit 无符号整型数  */
uint16_t /* unsigned 16-bit integer 16bit 无符号整型数 */
uint32_t /* unsigned 32-bit integer 32bit 无符号整型数 */
int8_t /* signed 8-bit integer       8bit 有符号整型数 */
int16_t   /* signed 16-bit integer  16bit 有符号整型数 */
int32_t   /* signed 32-bit integer  32bit 有符号整型数 */
float32_t /* 32-bit floating-point  32bit 浮点数 */
float64_t /* 64-bit floating-point  64bit 浮点数 */
```

非特定的对象名称命名时标识对象类型。 例如：

```
uint8_t u8a;     /* 8-bit unsigned integer 8bit 无符号整型数 */
int16_t s16b;    /* 16-bit signed integer  16bit 无符号整型数 */
int32_t s32c;    /* 32-bit signed integer  32bit 无符号整型数 */
bool_t bla;      /* essentially Boolean     基本类型为布尔型 */
enum atag ena;   /* enumerated type         枚举类型         */
char chb;        /* character                 字符           */
float32_t f32a; /* 32-bit floating-point   32bit 浮点数      */
float64_t f64b; /* 64-bit floating-point   64bit 浮点数      */
```

6.10 了解参考文献

如果准则来自一个或多个已发布的来源，则在准则之后的方括号中以 d 表示。 这有两个目的。 首先，希望获得对指南背后原理的更充分理解的读者可以建议特定来源(例如，在考虑偏差要求时)。 其次，关于ISO C 标准中描述的可移植性问题，来源形式提供了有关问题性质的更多信息。

如果准则有一个或多个参考文献，则在准则之后的方括号中以 d 表示。这有两个目的：首先，希望更充分理解准则背后原理的读者可以追溯特其源头(例如，才考虑偏差时)；其次，关于 ISO C 标准中描述的可移植性问题，参考文献提供了更多信息。

没有参考文献的规则可能源自制定公司的内部标准，或者由审核者建议，或者被广泛接受的良好做法。

6.10.1 ISO C 可移植性问题参考

ISO C 的可移植性问题在相关标准的附件小节中进行了描述。 重中之重，对于 C90 和 C99，参考文献均来自原始标准，而不是任何包含更正和修正的更高版本。 做出此决定的原因部分是历史原因，因为 MISRA C：2004 使用了该方案，部分是实际的原因是，尽管有草案[12]，但尚无正式的 C99 标准修订版纳入其技术勘误，[12]。

与此类参考文献对应的相关标准的小节为：

参考文献	ISO 9899:1990 附录G	ISO 9899:1999 附录J
未指定	G.1	J.1
未定义	G.2	J.2
实施	G.3	J.3
区域设置	G.4	J.4

参考文献后的文字具有以下含义：

- [2]的附件 G：该文本标识附件相关部分中一个或多个项目的编号，从该节的开头开始编号。 因

此，例如[Locale 2]指的是标准 G.4 节中的第二项，而[未指定 3，5]指的是标准 G.2 节中的第三和第五项。

- [8]的附件 J：对于 J.1，J.2 和 J.4 节，适用与上述相同的解释。

对于 J.3 节，该文本标识 J.3 的小节，后跟带括号的项目号。 因此，例如[未指定 6]指的是标准 J.1 节中的第六项，而[实施J3.4(2，5)]指的是标准 J.3.4 节中的第二和第五项。

如果准则基于本标准的附件 G(C90)或附件 J(C99)中的问题，则有助于读者理解未指定，未定义，实现和区域设置之间的区别。 这些将在这里简要解释，并且可以在 Hatton [3]中找到更多信息。

Unspecified 未指定

这些语言结构(*语句*)能够成功编译，但是编译器编写者在其中可以自由地进行构造。 一个例子是规则 13.2 中描述的“评估顺序”。

尽管不可避免地会使用某些未指定的行为，但假设编译器生成的目标代码以特定的方式运行是不明智的。 编译器甚至不需要在所有可能的构造上一致地执行。

MISRA C 会识别出可能导致不安全行为的未指定行为的实例。

Undefined 未定义

这些本质上是编程错误，但是编译器编写程序没有义务提供错误消息。 比如函数的无效参数，或者其参数与定义的参数不匹配的函数。

从安全的角度来看，这些尤为重要，因为它们代表了编译器不一定能捕获的编程错误。

Implementation 实施/实现

这些类似于“unspecified 未指定”的问题，主要区别在于编译器编写者必须采用一致的方法并对其进行记录。 换句话说，功能可能因一个编译器而异，从而使代码不可移植，但在任何一个编译器上，其行为都应得到明确定义。 一个示例是整数除法运算符和余数运算符/和%在应用于一个正整数和一个负整数时的行为。

从安全的角度来看，Implementation 定义的行为往往不太重要，前提是编译器编写者已充分记录了预期的方法，然后一致地实现了该方法。 建议尽可能避免这些问题。

Locale 区域设置

特定于区域设置的行为是一小部分功能，可能会随国际要求而变化。 这方面的一个示例是用“,”字符而不是“.”字符表示小数点。 本文档中没有解决由该来源引起的问题。

我们使用英文字符，不会出现这个类别的问题。

6.10.2 其他参考文献

ISO C 可移植性问题以外的参考文献均来自以下来源：

参考文献	源
MISRA Guidelines	The MISRA Guidelines[15]
Koenig	“C Traps and Pitfalls”, Koenig [31]
IEC 61508	IEC 61508:2010 [32]
ISO 26262	ISO 26262:2011 [23]
DO-178C	DO-178C [24]

除非另有说明，否则参考文献后的文字给出了相关的页码。

6 指令(Directives)

7.1 实施/实现(The implementation)

Dir 1.1	程序输出所依赖的任何由实现定义的行为都应被记录和理解

C90［附录 G.3］, C99［附录 J.3］

级别 必要

适用 C90，C99

展开

本文档的附录G 列出了 C90 和 C99 需要被记录和理解的实现定义的行为：

- 可能导致意想不到的程序行为；
- 即使符合 MISRA-C 所有其他准则，也不一定能完全规避。

所有这些行为，必须：

- 被记录；
- 被开发人员所了解。

小贴士：每一个实现定义的行为的记录，应有统一的文档。如果缺少任何文档，应该咨询实现的开发人员。

原理

知道一段程序的输出是有意而非偶然，非常重要。

下面介绍了一些与安全性相关的嵌入式软件可能依赖的更常见的实现定义的行为。

※核心行为

大多数程序可能需要执行定义的基本行为，包括：

- 如何识别编译过程中产生的诊断信息；
- 函数 main 的类型，在独立实现中通常声明为 void main (void)；*实际项目中，应根据芯片不同，进行不同的声明，一如文字中所写“通常”。*
- 标识符中的有效字符数 - 为 Rule 5.2 配置分析工具所需；
- 源代码和执行字符集；*理解为源代码文件的文档编码规则，如 ANSI/UTF-8?*
- 整型数的大小；
- #include 的名称如何映射到文件名并位于主机文件系统中

※语言扩展（*对语言规范的扩展，如前面提到的“//”和“bit”）*

在嵌入式系统中，语言扩展通常用于提供对外围设备的访问，以及将对象放入具有特殊属性(如 Flash EEPROM 或快速访问RAM)的内存区域。一个符合规范的实现被允许提供扩展，只要它们不改变任何严格符合规范的程序的含义。

某些 C90 实现可能会提供扩展，以实现 C99 功能的子集。

实现扩展的一些方法有：

- # pragma 预处理指令或_Pragma 操作符(仅 C99)；
- 新的关键字(keywords)。

※标准库

标准库实现的一些重要方面可能是：

- 使用某些标准库函数时赋值给 errno 的值；*errno 是记录系统的最后一次错误代码。详细信息可以网上搜索，它涉及调试程序的重要方法。*
- 时间和时间函数的实现；
- 文件系统的特性。

※应用程序二进制接口(The Application Binary Interface (ABI))

有时候我们会遇到必须将 C 代码与汇编代码混编的情况，如需要提高关键位置的执行速度时。也可能需要链接由不同编译器、不同语言生成的代码。

编译器的应用程序二进制接口(ABI)即是为这些任务提供所需的信息，包括必要定义的行为。它通常指定：

- 如何在寄存器中和堆栈上传递函数参数；
- 如何返回函数值；
- 哪些寄存器必须由函数保留；
- 具有自动存储期限的对象如何分配到堆栈帧；
- 每种数据类型的对齐要求；
- 如何布置结构以及如何将位字段赋值给存储单元。

一些处理器具有标准的 ABI，则所有实现都应使用它。 在没有标准 ABI 的地方，应提供自己的实现。

※整数除法

在 C90 中，其中一个操作数都为负值的有符号整数除法或余数运算可能向下或向零舍入。 在 C99 中，四舍五入可以保证接近零。***因此，我们必须明确工程中整数除法的舍入规则。***

※浮点实现

浮点类型的实现可能对程序行为产生重大影响，例如：

- 值的范围及其存储的精度；
- 浮点运算后的取整方向；
- 转换为较窄的浮点类型或整数类型时的取整方向；
- 下溢，溢和非数值存在的行为;(NaNs　　Not-a-Numbers)
- 库函数的定义域和值域错误的事件的行为。

参阅

Rule 5.1，Rule 5.2

7.2 编译与构建(Compilation and build)

Dir 2.1　　所有源文件都应通过编译且没有任何编译错误

级别　　必要

适用　　C90，C99

原理

有的编译器可能会尽管编译错误，但仍会生成目标模块。但是，执行生成的程序可能会产生意外的行为。

有关控制和分析编译器消息的更多信息，请参见第 5.3.1 节。

参阅

Rule 1.1

7.3 需求可追溯性(Requirements traceability)

Dir 3.1　　所有代码应可追溯到书面要求

[DO-178C Section 6.4.4.3.d]

级别　　必要

适用　　C90，C99

原理

不需要满足项目要求的功能会导致不必要的路径。 软件开发人员可能没有意识到此附加功能可能带来的广泛影响。 例如，开发人员可能会添加代码，以在每次到达程序中的特定点时切换处理器输出引脚的状

态。 在开发过程中测量时序或触发仿真器或逻辑分析器时，这可能非常有用。 但是，即使由于软件要求说明书中未提及该引脚而似乎未使用该引脚，也可能将其连接至目标控制器中的执行器，从而导致不良的外部影响。

追溯代码到文件化要求的方法应由项目决定。 一种实现可追溯性的方法是对照相应的已针对需求进行审核过的设计文档来审核代码。

小贴士：本准则与保护性编码策略的提供之间不应有任何冲突，因为后者应是关键系统要求的一部分。

7.4 代码设计(Code Design)

Dir 4.1 运行时的故障，应尽量减少

C90 [Undefined 15, 19, 26, 30, 31, 32, 94]

C99 [Undefined 15, 16, 33, 40, 43 - 45, 48, 49, 113]

级 别 必 要

适 用 C90， C99

原理

C 语言设计提供非常有限的内置的运行时检查。虽然这种方法可以生成紧凑而快速的可执行代码，但是它将运行时检查的负担放在了程序员身上。因此，为了达到预期的健壮性水平，程序员必须仔细考虑在可能发生运行时错误的地方添加动态检查。

有时可以证明操作数的值排除了在计算表达式时发生运行时错误的可能性。在这种情况下，不需要进行动态检查，前提是支持其省略的参数被记录在案。任何这类文件都应包括论点所依据的假设。这些信息可以在代码的后续修改中使用，以确保参数保持有效。

用于最小化运行时故障的技术应有计划和被记录，例如记录在设计标准，测试计划，静态分析配置文件和代码复查清单中。 这些技术的性质可能取决于项目的完整性要求。 有关更多详细信息，请参见第 5.2 节。

小贴士：如果存在运行时错误，则表示违反Rule 1.3。

下列说明对需要考虑提供动态检查的领域提供了一些指导：

- 算术错误：这包括在表达式求值中出现的错误，例如上溢，下溢，被零除或因移位而丢失有效位。 在考虑整数溢出时，请注意，无符号整数计算不会严格溢出，而是会产生确定的值，但可能是意料之外的值。 应仔细考虑算术表达式中的值范围和运算顺序，例如：

```
float32_t f1 = 1E38f;
float32_t f2 = 10.0f;
float32_t f3 = 0.1f;
```

```
float32_t f4 = ( f1 * f2 ) * f3; /* (f1 * f2) 将溢出 */
float32_t f5 = f1 * ( f2 * f3 ); /* 不会溢出，因为(f2*f3)(约)等于 1 */
if ( ( f3 >= 0.0f ) && ( f3 <= 1.0f ) )
{
/* 不会溢出，因为 f3 已被确知其值域为[0...1]，
 * 所以乘法运算的结果会符合类型 float32_t */
    f4 = f3 * 100.0f;
}
```

- 指针算术：确保在动态计算地址时，计算出的地址是合理的，并且指向有意义的地方。特别需要确保的是，如果一个指针指向数组中的某个点，那么当指针被增加或以其他方式改变时，它仍然指向同一个数组中的某个点。参见指针算法的限制—Rule 18.1、Rule 18.2 和 Rule 18.3。
- 数组超限错误：在使用数组索引对数组进行索引之前，请确保数组索引在数组大小的范围内—Rule 18.1。
- 函数参数：在将参数传递给库函数(Dir 4.11)之前，应检查其有效性。
- 指针引用：除非指针已知为非 NULL，否则在引用指针之前应进行运行时检查。一旦进行了检查，在单个函数中推断指针是否已经更改以及是否需要另一个检查就相对简单了。跨函数边界进行推理要困难得多，特别是在调用其他源文件或库中定义的函数时。

```
/*
 * 函数功能：获取消息正文的指针，检查消息头并返回指定消息正文的指针，若消息无效，则返回 NULL
 */
const char *msg_body ( const char *msg )
{
    const char *body = NULL;
    if ( msg != NULL )
    {
        if ( msg_header_valid ( msg ) )
        {
            body = &msg[ MSG_HEADER_SIZE ];
        }
    }
    return body;
}
char msg_buffer[ MAX_MSG_SIZE ];
const char *payload;
payload = msg_body ( msg_buffer );
/* Check if there is a payload */
if ( payload != NULL )
{
    /* 处理 payload */
}
```

- dynamic memory 动态存储：如果正在执行动态内存分配，则必须检查每个分配是否成功，并且设计并测试了适当的降级或恢复策略。

参阅

Dir 4.11, Dir 4.12, Rule 1.3, Rule 18.1, Rule 18.2, Rule 18.3

Dir 4.2 汇编语言的所有运用都应记录在案

级别 建议

适用 C90，C99

展开

应该记录使用汇编语言的基本原理以及 C 和汇编语言之间的接口连接机制。

原理

汇编语言代码是由具体实现定义的，因此不可移植。

Dir 4.3 汇编语言应被封装和隔离

级别 必要

适用 C90，C99

展开

使用汇编语言说明的地方，应将它们封装和隔离在下述结构中：

- 汇编语言函数；
- C 语言函数(C99 首选内联函数)；
- C 语言宏。

原理

出于执行效率的考虑，有时必须嵌入简单的汇编指令，例如启用/禁用中断。若有必要(嵌入汇编指令)， 则建议使用宏或内联函数(C99)来实现。

封装汇编语言的益处：

- 提高可读性；
- 封装宏或函数的名称和文档，清楚地说明了汇编指令的意图；
- 所有执行相同指定功能的汇编语言代码可以共享相同的封装，从而提高可维护性；

- 当实现方案或静态分析变动时，封装后的汇编语言可以轻松的被替换。

小贴士：内联汇编语言的使用是对标准C 的扩展，因此它违反了 Rule 1.2。

示例

```
#define NOP asm(" NOP")
```

Dir 4.4　　代码段不应被“注释掉”

级别　　建议

适用　　C90，C99

展开

此规则适用于“//”和“/*…*/”注释样式。

原理

如果要求源代码中的某部分不被编译，应使用条件编译来实现(即：带注释的“#if”或“#ifdef”结构)。使用注释标记符的方式来实现相同目的则很危险，因为 C 语言不支持嵌套注释，所以存在于代码部分中的任何注释都会改变效果。

参阅

Rule 3.1，Rule 3.2

Dir 4.5　　具有相同可见性的相同名称空间中的标识符在印刷/屏幕显示上应明确

级别　　建议

适用　　C90，C99

展开

“明确”一词的定义应根据编写源代码所用的字母表和语言来确定。

对于英语单词中使用的拉丁字母，建议标识符至少应能明确区分下述组合的不同:

- 小写字符与其等效的大写字符的交换，如 X，x；W，w；V，v；Z，z 等；
- 下划线字符存在与否；
- 字母“ O”和数字“ 0”的互换；
- 字母“ I”和数字“ 1”的互换；
- 字母“ I”和字母“ l”(el)的互换；

- 字母“ l”(el)和数字“ 1”的互换；
- 字母“ S”和数字“ 5”的互换；
- 字母“ Z”和数字“ 2”的互换；
- 字母“ n”和字母“ h”的互换；
- 字母“ B”和数字“ 8”的互换；
- 字母序列“ rn”(“ r”后跟“ n”)和字母“ m”的互换；

原理

根据使用显示字体的不同，即使字符不同，某些字形也可能看起来相同。这可能会导致开发人员将不同标识符混淆。

因为使用字体的不同，会出现上面的所述的相似字符混淆的情况，因此，我们应避免上述会产生显示歧义的标识符命名。

示例

下面的例子假设了用于拉丁文字母表和英语语言的**展开**的解释(*即容易混淆的命名*)：

```
int32_t id1_a_b_c;
int32_t id1_abc;    /* Non-compliant */
int32_t id2_abc;
int32_t id2_ABC;    /* Non- compliant */
int32_t id3_a_bc;
int32_t id3_ab_c;   /* Non-compliant */
int32_t id4_I;
int32_t id4_1;      /* Non-compliant */
int32_t id5_Z;
int32_t id5_2;      /* Non-compliant */
int32_t id6_O;
int32_t id6_0;      /* Non-compliant */
int32_t id7_B;
int32_t id7_8;      /* Non-compliant */
int32_t id8_rn;
int32_t id8_m;      /* Non-compliant */
int32_t id9_rn;
struct
{
    int32_t id9_m; /* Compliant(这里合规，是因为它和上面的 id9_rn 不是同一命名空间) */
};
```

Dir 4.6 应使用指示大小和符号的 typedef 类型代替基本数字类型

级别 建议

适用　　C90，C99

展开

基本数字类型(char，short，int，long，long long(C99)，float，double 和 long double)不应被使用，而应使用由 typedef 定义的特定长度的数字类型。

对于 C99，应使用<stdint.h>里提供的类型。 对于 C90，应定义和使用等效的类型。

定义为特定长度的类型，其实现必须确实具有该长度。

在位域(或名位带)的声明中，不必使用 typedef 定义的类型。(实际上位域要求是不应使用 typedef 类型来定义)

例如，在 32 位 C90 实现上，以下定义可能适用：

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

原理

在分配的内存量很重要的情况下，使用特定长度类型可以清楚地确认为每个对象保留多少存储空间。遵守此准则并不能保证可移植性，因为 int 类型的大小可以确定表达式是否要进行整数提升。 例如，如果使用 16 位实现 int，则不会提升类型为 int16_t 的表达式，但是如果使用 32 位实现 int，则将提升类型为 int16_t 的表达式。 附录 C 中有关整数提升的部分将对此进行详细讨论。

小贴士：在存储要求和可移植性方面，定义大小与实现的类型不同的特定长度类型会适得其反。 应注意避免定义尺寸错误的类型。

如果抽象类型是按照特定长度类型定义的，那么这些抽象类型就没有必要指定大小或符号，甚至可能不希望这样。例如，下面的代码定义了一种抽象类型，以公斤表示质量，但不表示其大小或符号：

```
typedef uint16_t mass_kg_t;
```

与标准库或项目控制范围之外的代码交互时，可能不希望应用此准则。

例外

1. 可以在 typedef 中使用基本的数值类型来定义特定长度类型。
2. main 函数可以使用 int 而不是 typedef 类型作为返回类型，即允许 int main (void)。

3. main 函数输入参数 argc 可以使用 int 而不是 typedef 类型。

4. main 函数输入参数 argv 可以使用 char 而不是 typedef 类型。

示例

```
/* 违规 - “int”被用于直接定义一个变量对象    */
int x = 0;
/* 合规 - “int”被用于定义特定长度的数据类型 */
typedef int SINT_16;
/* 违规 - 未指定符号和长度 */
typedef int speed_t;
/* 合规 - 对已规定有符号和长度的类型进行进一步抽象，不需要重复定义其符号和长度 */
typedef int16_t torque_t;
```

定义计量单位，可以不规定其数据长度，但定义定性概念，如例子中的速度，则需要定义其数据长度。

Dir 4.7　　如果函数返回错误信息，则应测试该错误信息

级别　　必要

适用　　C90，C99

展开

视为返回错误信息的功能列表应由项目确定。

函数返回的错误信息应以有意义的方式进行测试。

原理

一个函数(无论它是标准库的一部分，第三方库还是用户定义的功能)都可以视为提供了一些指示错误发生的方法。 这可以通过错误标志，某些特殊的返回值或其他方式进行。 每当函数提供这种机制时，调用程序应在函数返回后立即检查错误指示。

但是，请注意，检查函数的输入值被认为是比在函数完成后尝试检测错误更可靠的错误预防方法(请参见 Dir 4.11)。

例外

如果可以(例如通过检查参数)显示函数不能返回错误指示，则无需执行检查。

参阅

Dir 4.11, Rule 17.7

Dir 4.8 如果一个指向结构体或联合体的指针在编译/解释时从未被反引用，则应隐藏该对象的实现

级别 建议

适用 C90，C99

展开

对象的实现应通过指向不完整类型的指针隐藏。

原理

如果从未取消引用(be dereferenced)结构或联合的指针，则不需要该对象的实现详细信息，并且应保护其内容免受意外更改。取消引用(dereferenced)，即访问指针指向的对象。

隐藏实现细节将创建一个不透明类型，可以通过指针对其进行引用，但是其内容可能无法访问。

示例

```
/* Opaque.h */
#ifndef OPAQUE_H
#define OPAQUE_H
typedef struct OpaqueType *pOpaqueType;
#endif

/* Opaque.c */
#include "Opaque.h"
struct OpaqueType
{
    /* 对象的实现 */
};

/* UseOpaque.c */
#include "Opaque.h"
void f ( void )
{
    pOpaqueType pObject;
    pObject = GetObject ( ); /* 获取 OpaqueType 对象的句柄 */
    UseObject ( pObject );    /* Use it... */
}
```

理解：如果一个结构体或联合体类型，在外部使用时，仅有其指针操作，从不会访问其类型的内部成员，则应如示例一样，将类型的具体实现(成员)定义在 c 文件中，而在头文件中仅提供该指针类型的定义。这种定义方法定义的指针，即为不完整类型的指针，或叫不透明类型的指针，如上面所述，外部无法访问该指针指向类型的真实数据。

Dir 4.9　应该优先使用函数，而不是类似函数的宏(如果它们可以互换)

[Koenig 78 - 81]

级别　建议

适用　C90，C99

展开

本指南仅适用于语言标准的语法和约束允许的功能。

原理

在大多数情况下，应使用函数代替宏。 函数执行参数类型检查并评估其参数一次，从而避免了潜在的多重副作用的问题。 在许多调试系统中，单步执行功能比宏要容易得多。尽管如此，在某些情况下宏还是有用的。

在决定使用函数还是宏时应考虑下述因素：

- 函数参数和结果类型检查的好处；
- C99 中内联函数的可用性，但是请注意，内联所作用的程度是实现定义的；
- 在代码大小和执行速度之间进行权衡；
- 编译时检查的可能性是否重要：具有常数参数的宏比相应的函数调用更可能在编译时检查；即：有些常数参数的宏在编译时会直接替换为宏的计算结果，这类宏更适合保留其宏形式定义。
- 参数对函数是否有效：宏参数为文本，而函数参数为表达式；
- 易于理解和可维护性。

示例

以下示例符合标准。 这些类似函数的宏不能用函数替换，因为它具有 C 运算符作为参数：

```
#define EVAL_BINOP( OP, L, R ) ( ( L ) OP ( R ) )
uint32_t x = EVAL_BINOP ( +, 1, 2 );
```

在下面的示例中，使用宏初始化具有静态存储持续时间的对象是合规的，因为此处不允许进行函数调用。

```
#define DIV2(X) ( ( X ) / 2 )
void f ( void )
{
    static uint16_t x = DIV2 ( 10 ); /* 合规 - 此处不允许函数调用 */
           uint16_t y = DIV2 ( 10 ); /* 违规 - 此处允许函数调用   */
}
```

参阅

Rule 13.2, Rule 20 .7

Dir 4.10　应采取预防措施以防止头文件的内容被多次包含

[Koenig 78－81]

级别　必要

适用　C90，C99

原理

当翻译单元包含嵌套头文件的复杂层次结构时，可能不止一次包含特定头文件。这在最好情况下可能是一种混淆的根源。如果这个多重包含导致多个或冲突的定义，那么这会导致未定义的或错误的行为。

示例

```
/* file.h */
#ifndef FILE_H
/* 违规 - 缺少 #define FILE_H */
#endif
```

为了便于检查，应使用下列两种形式之一，对头文件的内容进行保护，避免多次包含。

```
<文件开始>
#if !defined ( identifier )
#define identifier
    /* 文件内容 */
#endif
<文件结束>

< 文 件 开 始 >
#ifndef identifier
#define identifier
    /* 文件内容 */
#endif
<文件结束>
```

小贴士：*用于测试和记录是否已包含给定头文件的标识符在项目中的所有头文件中应是唯一的。*

小贴士：*这些格式中的任何地方都允许注释。*

Dir 4.11　应检查传递给库函数的值的有效性

C90 [Undefined 60, 63, 96; Implementation 45－47]

C99 [Unspecified 30, 31, 44, 48－50;

Undefined 102, 103, 107, 112, 180, 181, 183, 187, 189;

Implementation J.3(8－11)]

级别　必要

适用　C90，C99

展开

项目的性质和组织将决定哪些库以及这些库中的功能应受本指令的约束。

原理

标准库中的许多函数都不检查传递给它们的参数的有效性。而且，即使标准中要求进行检查，或编译器作者声称检查参数的情况下，也不能保证进行充分的检查。

类似地，其他库中的函数的接口描述可能不会指定这些函数执行的检查。还有一个风险是，指定的检查不一定充分执行。

程序员应该为所有具有受限输入域(标准库、第三方库和内部库)的库函数提供适当的输入值检查。

具有限制域和需要检查的标准库功能的示例如下:

- ◆ <math.h>中的许多数学函数，如:
  - — 负数不能传递给 sqrt 或 log 函数;
  - — fmod 函数的第二个参数不能为 0;
- ◆ 当函数 toupper 传递一个非小写字母的参数(tolower 也有一样的问题)时，某些实现可能会产生意外结果。
- ◆ 如果传递了无效值，<ctype.h>中的字符测试函数将显示未定义的行为;
- ◆ 应用于最负整数的abs 函数给出未定义的行为。【最大负整数？】

尽管<math.h>中的大多数数学库函数都定义了允许输入的域，但是当发生域错误时它们返回的值可能因编译器而异。 因此，预检查输入值的有效性对于这些功能特别重要。

程序员应明确那些应该合理地应用于正在使用的功能的任何领域约束(可能在界面描述中记录或可能未记录)，并提供适当的检查以确保输入值位于该域内。 当然，如果需要，可以通过知道参数代表什么以及什么构成参数的合理范围的值来进一步限制该值域。

我们可以通过多种方式来满足本准则的要求，包括:

- 在调用函数前进行检查；
- 检查所调用的库函数内的值，这特别适用于内部设计的库，当然，若供应商能够证明他们提供的函数库内也具备该检查功能，这种方法也可以适用于该外部库；
- 生成具备执行检查的功能的“包装”版本，然后调用其原始功能；【即做检查封装】
- 静态证明输入参数绝不能采用无效值。

参阅

Dir 4.1, Dir 4.7

Dir 4.12 不得使用动态内存分配

级别 必要

适用 C90，C99

展开

此规则适用于所有的动态内存分配的封装，包括：

- 标准库提供的内容；
- 第三方软件包。

原理

标准库的动态内存分配和取消分配例程可能导致 Rule 21.3 中所述的不确定行为。 任何其他动态内存分配系统都可能表现出与标准库相似的未定义行为。

我们必须检查第三方例程的规范，以确保不会无意中使用动态内存分配。

如果决定使用动态内存，则应注意确保软件以可预测的方式运行。 例如，存在以下风险：

- 可能没有足够的内存来满足请求-必须注意确保对分配失败有安全适当的响应；
- 根据使用方式和碎片的程度，执行分配或取消分配所需的执行时间差异很大。

示例

为了方便起见，这些示例基于标准库的动态存储功能的使用，因为它们的接口是众所周知的。

在此示例中，在第一次调用 free 之后，行为是不确定的，因为指针 p 的值变得不确定。 尽管在调用 free 之后指针中存储的值保持不变，但是在某些目标上，指针所指向的内存可能不再存在，并且复制该指针的行为可能导致内存异常。

```
#include <stdlib.h>
void f ( void )
{
    char *p = ( char * ) malloc ( 10 );
    char *q;
```

```
    free ( p );
    q = p;          /* 未定义的行为 - 变量 p 的值不确定 */

    p = ( char * ) malloc ( 20 );
    free ( p );
    p = NULL;       /* 给被释放空间的指针赋值 NULL，以保证其值确定 */
}
```

参阅

Dir 4.1, Rule 18.7, Rule 21.3, Rule 22.1, Rule 22.2

Dir 4.13 用于对资源进行操作的功能应按适当的顺序调用

级别 建议

适用 C90，C99

展开

对资源的操作的一组函数通常具有三种操作：

1. 资源的分配，例如打开文件；
2. 资源的释放，例如关闭文件；
3. 其他的操作，例如从文件读取。

对于每个这样的功能集，其操作的所有调用都应按适当的顺序进行。

原理

静态分析工具能够提供路径分析检查功能，该功能可以识别未正确调用释放功能函数的调用路径。为了使这种自动检查的好处最大化，鼓励开发人员通过设计并向静态分析器声明平衡功能集来启用这些检查。

示例

```
/* 这些函数旨在配对 */
extern mutex_t mutex_lock ( void );
extern void mutex_unlock ( mutex_t m );

extern int16_t x;

void f ( void )
{
    mutex_t m = mutex_lock();

    if ( x > 0 )
    {
```

```
        mutex_unlock ( m );
    }
    else
    {
        /* Mutex 未在此路径上解锁 */
    }
}
```

参阅

Rule 22.1, Rule 22.2, Rule 22.6

8 规则(Rules)

8.1 标准 C 环境(A standard C environment)

Rule 1.1 程序不得违反标准 C 语法和约束，并且不得超出具体实现的编译限制

[MISRA G uidelines Table 3], [IEC 61508-7: Table C.1], [ISO 26262-6: Table 1]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

程序应仅使用所选标准版本中指定的 C 语言及其库的功能(请参阅第 3.1 节)。

该标准允许实现提供语言扩展，并且该规则允许使用此类扩展。

除非使用语言扩展，否则程序不得有以下行为：

- 包含任何违反标准中描述的语言语法的内容；
- 包含任何违反标准规定约束条件的内容。

程序不得超过实施所规定的翻译限制。 最低翻译限制由标准规定，但实现可能会提供更高的限制。

小贴士：符合标准的实现会生成语法和约束违规的诊断，但请注意：

- *诊断不一定是报错，也可以是告警；*
- *尽管存在语法或约束冲突，也可能编译程序成功并生成可执行文件。*

小贴士：如果超出编译限制，则符合标准的实现无需生成诊断； 可执行文件可能会生成，但不能保证正确执行。

原理

在制定这些准则期间，并未考虑与受支持的 ISO / IEC 9899 版本之外的语言功能相关的问题。

某些违规的实施无法被诊断，例如，[38] p135，示例 2，标题为“写入到常量区的错误”。**示例**

一些 C90 编译器使用_inline 关键字提供对内联函数的支持。 这种情况下，使用__inline 的 C90 程序将符合此规则，条件是打算使用此类编译器进行编译。

许多用于嵌入式目标的编译器提供了其他关键字，这些关键字用对象所在的存储区的属性来限定对象类型，例如：

- ____zpage：可以使用简短指令访问对象
- ____near：指向对象的指针可以保留 16 位
- ____far：指向对象的指针可以保留 24 位

如果编译器支持这些关键字作为语言扩展，则使用这些其他关键字的程序将符合此规则。

参阅

Dir 2.1, Rule 1.2

Rule 1.2　不应该使用语言扩展

[MISRA G uidelines Table 3], [IEC 61508-7: Table C.1], [ISO 26262-6: Table 1]

级别　建议

分析　不可判定，单一编译单元

适用　C90，C99

原理

依赖语言扩展的程序的可移植性较不依赖语言扩展的程序低。尽管本标准要求了符合标准的实施文档要提供语言的所有扩展，但仍存在无法完整描述所有情况的风险。

如果不使用此规则，则应在项目的设计文档中明确说明每种被使用的语言扩展。且应记录每个扩展的有效使用方法，如编译检测方法和诊断方法。

人们已经认识到，在嵌入式系统中使用语言扩展很有必要。但本标准要求语言的扩展不能改变任何严格符合程序的行为。例如，作为一项语言扩展功能，编译器可能会实施对二进制逻辑运算符的全面评估，尽管标准规定了一旦确定结果就立即停止评估。这样的扩展，就不符合标准，因为在逻辑与运算符的右操作数中总会产生副作用，从而产生非预期的行为。

【举例说明 ： if ((NULL != FuncPointer) && (*FuncPointer())，这样的语句是符合语法的，且就是利用了“一旦确定结果立即停止评估”的特性，在 FuncPointer 值为 NULL 时执行“&&”的右操作数会非

常危险，程序会跑到哪完全不可预知】

参　　阅

Rule 1.1

Rule 1.3　不得发生未定义或严重的未指定行为

级别　必要

分析　不可判定，系统范围

适用　C90，C99

展开

一些未定义或未指定的行为有特定的规则处理。此规则意在防止其他未定义和关键的未指定行为。附录 H 列出了这些未定义和被认为是关键的未指定的行为。

原理

引起未定义或未指定行为的任何程序都可能无法以预期的方式运行。 在许多情况下，其结果仅仅是使程序不可移植，但也可能发生更严重的问题。 例如，未定义的行为可能会影响计算结果。 如果软件的正确执行取决于此计算，则可能会损害系统安全性。 如果未定义的行为仅在极少数情况下才表现出来，则该问题将尤其难以检测到。

MISRA C 的许多准则旨在避免某些未定义和未指定的行为。 例如，遵守 Rule 11.4、Rule 11.8 和 Rule 19.2 的所有内容可确保在C 中不能创建指向使用 const 限定类型声明的对象的非 const 限定指针。这避免了 C90 [Undefined 39]和 C99 [Undefined 61]。 但是，其他行为不受特定准则的限制，因为：

- 不太可能遇到该行为；
- 除了显示给出避免行为的描述外，无法提供实际的操作指导。

MISRA C 并没有将所有的未定义和未指定的行为都引入进来，而是明确定位了那些被认为最重要且最可能在实践中发生的行为。本规则涵盖了所有那些没有被特定准则定义的行为。附录 H 列出了所有的未定义和未指定行为，以及防止其发生的对应 MISRA C 准则。亦即是说，它规定了本准则和其他准则的涵盖范围。

小贴士：某些实现可能为标准中列出的一些未定义和未指定的行为提供定义良好的行为。 如果依靠这样定义良好的行为(包括通过语言扩展)，则有必要针对这些行为背离此规则。

参　　阅

Rule 4.1

8.2 未使用的代码(Unused code)

Rule 2.1 项目不得包含不能访问的代码(unreachable code)

[MISRA G uidelines Table 3], [IEC 61508-7: Table C.1], [ISO 26262-6: Table 1]

级别 必要

分析 不可判定，单一编译单元

适用 C90，C99

原理

如果程序没有表现出任何未定义的行为，则无法执行不可达代码，并且不会对程序的输出产生任何影响。 因此，存在不可达代码可以表示存在程序逻辑错误。

一般编译器会在编译时移除所有不可达代码，尽管它并非必须这么做。而那些无法被编译器移除的不可达代码将造成资源浪费，比如：

- 它占用目标设备的内存空间；
- 它的存在可能导致编译器在围绕不可达代码进行控制权转移时选择更长、更慢的跳转指令；
- 在循环中，它可能会阻止整个循环驻留在指令缓存中。

有时，人们会主动插入一段看起来不可达的代码，用以处理特殊情况。例如，在 switch 语句中，控制表达式的每个可能值都由显式情况描述，而 default 子句应根据规则 16.4 出现。default 子句的用途是捕获通常不应该出现但可能由于以下原因而生成的值：

- 程序中存在未定义的行为；
- 处理器硬件故障。

如果编译器能够确认 default 子句不可达，则可以将其删除，从而清除该防御措施。如果该防御措施非常重要，则下述两者必须满足其一：要么保证编译器在证实不可达时仍保留该代码，要么采取措施使防御代码可达。前者与本规则有偏差，可能需要检查目标代码或使用单元测试来保证这种偏差。后者通常可以通过易失性(volatile)访问来实现。举例来说，编译器一般在执行 switch 语句前，已经确定了其条件 x 的值是否被 case 子句所覆盖，如：

```
uint16_t x;
switch (x)
```

而通过强制将x 识别为易失性左值的方式来访问它，编译器就必须假定这个表达式可以有任何值：

```
switch ( *( volatile uint16_t * ) &x )
```

小贴士：通过预编译指令排除的代码不受本规则约束，因为它们不会在后续编译阶段显示。

示例

```
enum light { red, amber, red_amber, green };
enum light next_light ( enum light c )
{
    enum light res;
    switch ( c )
    {
        case red:
            res = red_amber;
            break;
        case red_amber:
            res = green;
            break;
        case green:
            res = amber;
            break;
        case amber:
            res = red;
            break;
        default:
        {
            /* 当参数 c 的值不是枚举型 light 的成员时, 此 default 分支才可达 */
            error_handler ( );
            break;
        }
    }
    return res;
    res = c; /* 违规 - 此语句肯定不可达 */
}
```

参阅

Rule 14.3 16.4

Rule 2.2　不得有无效代码(dead code)

[IEC 61508-7 Section C.5.10], [ISO 26262-6 Section 9.4.5], [DO-178C Section 6.4.4.3.c]

级别　必要

分析　不可判定，系统范围

适用　C90，C99

展开

不论保留其执行还是被删除，均不会影响影响程序行为的操作，即构成无效代码。此处，我们假定语言扩展所引用的操作始终会对程序行为产生影响。

小贴士：在嵌入式系统中，通常情况下程序的行为不仅取决于它们的执行性质，还取决于它们发生的时间。

小贴士：不可达代码不是无效代码，因为它们不会被执行。不可达代码与无效代码的差别：不可达代码不会被执行，无效代码会被执行。

原理

无效代码的存在，可能表明程序逻辑中存在错误。由于无效代码可能会被编译器删除，因此它可能会引起混乱。

例外

强制转换为 void 是显示声明有意不使用的值。因此，强制转换本身不是无效代码。它被视为使用了操作数的值，所以它不被定性为无效代码。

示例

在此示例中，假定p 指向的对象在其他函数中使用。

```
extern volatile uint16_t v;
extern char *p;
void f ( void )
{
    uint16_t x;
    ( void ) v;         /* 合规 - v 虽有副作用，但其可被访问，且强转 void 是被例外
                         * 允许的                                                  */
    ( int32_t ) v;      /* 违规 - 执行无效                                         */
    v >> 3;             /* 违规 - ">>"运算无效                                     */
    x = 3;              /* 违规 - "="运算无效，x 随后未被读取                      */
    *p++;               /* 违规 - "*"运算的结果未被使用                            */
    ( *p )++;           /* 合规 - *p 自增                                          */
}
```

在下面的合规的示例中，_asm 关键字是语言扩展，而不是函数调用操作，因此不是无效代码。

```
__asm ( "NOP" );
```

在下面的示例中，函数 g 不包含无效代码，且其本身也不是无效代码，因为它不含任何操作。但是对它的调用无效，因为删除它不影响程序行为。

```
void g(void)
{
    /* 合规 - 此函数中无任何操作 */
}
void h(void)
{
    g(); /* 违规 - 该调用可以被移除 */
}
```

亦可查阅

Rule 17.7

Rule 2.3　项目不应包含未被使用的类型(type)声明

级别　建议

分析　可判定，系统范围

适用　C90，C99

原理

如果一个类型被声明但从未被使用过，对于审阅者来说，无法确定该声明是多余的还是被错误闲置的。错误闲置：本该被使用，却因错误而未被使用。

示例

```
int16_t unusedtype(void)
{
    typedef int16_t local_Type; /* 违规 */
    return 67;
}
```

Rule 2.4　项目不应包含未被使用的类型标签(tag)声明

级别　建议

分析　可判定，系统范围

适用　C90，C99

原理

如果一个类型标签被声明但从未被使用过，对于审阅者来说，无法确定该类型标签是多余的还是被错误闲置的。

示例

在下面的示例中，类型标签 record_t 仅在 record1_t 的类型声明中使用，而在需要使用该类型的位置均使用了 record1_t。此时，我们可以以省略标签的方式声明类型以满足本规则要求，如 record2_t。

```
typedef struct record_t /* 违规 */
{
    uint16_t key;
    uint16_t val;
} record1_t;
typedef struct          /* 合规 */
{
    uint16_t key;
    uint16_t val;
} record2_t;
```

```
void unusedtag(void)
{
    enum state { S_init, S_run, S_sleep }; /* 违规 */
}
```

Rule 2.5 项目不应包含未被使用的宏(macro)声明

级别 建议

分析 可判定，系统范围

适用 C90，C99

原理

如果一个宏被声明但从未被使用过，对于审阅者来说，无法确定该宏是多余的还是被错误闲置的。

示例

```
void use_macro(void)
{
    #define SIZE 4
    #define DATA 3      /* 违规 - DATA 未被使用 */
    use_int16(SIZE);
}
```

Rule 2.6 函数不应包含未被使用的执行标签(label)声明

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

原理

如果一个执行标签(label)被声明但从未被使用过，对于审阅者来说，无法确定该执行标签是多余的还是被错误闲置的。

tag 和 label，两者翻译为中文都是标签，差别在于 tag 为枚举、结构体、联合体类型的标签，label 为goto 语句执行目的地的标签，本文中为区分，将它们分别描述为了类型标签与执行标签。

示例

```
void unused_label(void)
{
    int16_t x = 6;
label1:                 /* 违规 */
    use_int16(x);
}
```

Rule 2.7 函数中不应有未使用的变量

级别　　建议

分析　　可判定，单一编译单元

适用 C90，C99

原理

绝大多数函数都将使用它们所定义的每一个参数。如果函数中的参数未被使用，则可能函数的实现与其预期定义不匹配。本规则强化描述了这一潜在的不匹配。

示例

```
void withunusedpara(uint1 6_t *para1, int16_t unusedpara) /* 违规 - 参数未使用 */
{
    *para1 = 42U;
}
```

8.3 注释(Comments)

Rule 3.1 注释中不得使用字符序列 /* 和 //

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

“/*”和“//”均为注释起始的字符序列，如果在一段由“/*”起始的注释中，又出现了“/*”或“//”，那么很可能是由缺少“*/”引起的。

如果这两个注释起始的字符序列出现在由“//”起始的注释中，则很可能是因为使用“//”注释掉了代码。

例外

“//”起始的注释里，允许再次出现“//”。

示例

参考下面代码片段：

```
/* some comment, end comment marker accidentally omitted
<<New Page>>
Perform_Critical_Safety_Function(X);
/* this comment is non-compliant */
```

在查看包含该函数调用的页面时，假定它是应执行的代码。由于意外删除了结束注释标记，因此不会执行对安全关键功能的调用。

在下面 C99 代码的示例中，“//”的出现改变了程序的含义：

```
x = y // /*
    + z
    // */
    ;
```

此示例得出的结果是 x=y+z，但在没有两个“//”的情况下，结果是 x=y。

参阅

Dir 4.4

Rule 3.2 “//”注释中不得使用换行(即“//”注释中不得使用行拼接符“\”)

级别 必要

分析 可判定，单一编译单元

适用 C99

展开

原文：Line-splicing occurs when the \ character is immediately followed by a new-line character. If the source file contains multibyte characters, they are converted to the source character set before any splicing occurs.

直译：当“\”字符后紧跟换行符时，将发生行拼接。 如果源文件包含多字节字符，则在进行任何拼接之前，它们将被转换为源字符集。这里旨在描述“\”在行末(换行符)的作用：它会将两行连接拼合，作为一行处理。

原理

如果包含“//”注释的源代码行在源字符集中以“\”字符结尾，则下一行将成为注释的一部分。 这可能会导致意外删除代码。

小贴士： *行拼接在 C90 和 C99 的第 5.1.1.2(2)节进行了介绍。*

示例

在下面的违规示例中，包含 if 关键字的物理行在逻辑上是前一行的一部分，因此是注释。

```
extern bool_t b;
void f(void)
{
    uint16_t x = 0; // comment \
    if (b)
    {
        ++x; /* if 语句被作为注释处理，这里无条件执行 */
    }
}
```

参 阅

Dir 4.4

8.4 字符集和词汇约定(Character sets and lexical conventions)

Rule 4.1 八进制和十六进制转译序列应有明确的终止识别标识

C90 [Implementation 11], C99 [Implementation J.3.4(7, 8)]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

八进制或十六进制转义序列应通过以下任一方式来标识其已终止：

- 另一个转义序列的开始，或者
- 字符常量的结尾或字符串的结尾。

原理

若八进制或十六进制转译序列后跟随其他字符，会造成混淆。例如，字符串“\x1f”仅由一个字符组成，而字符串“\x1g”则是由两个字符“\x1”和“g”组成。多字符常量表示为整数的方式是实现定义的(即不可取消或剔除)。

如果给字符常量或字符串文字中的每个八进制或十六进制转义序列增加显示的终止标识，则可以减少混淆的可能性。

示例

在此示例中，由 s1，s2 和 s3 指向的每个字符串都等效于字符串“Ag”。

```
const char *s1 = "\x41g";       /* 违规 - 无法区分哪个是转译序列，哪个又是普通字符   */
const char *s2 = "\x41" "g";    /* 合规 - 以字符串结束标识转译序列的结束             */
const char *s3 = "\x41\x67";    /* 合规 - 以新的转译序列起始标识前一个转译序列的结束 */
int c1 = '\141t';               /* 违规 - 无法区分哪个是转译序列，哪个又是普通字符   */
int c2 = '\141\t';              /* 合规 - 以新的转译序列起始标识前一个转译序列的结束 */
```

参阅

C90：6.1.3.4 节；C99：6.4.4.4 节

建议：不使用八进制和十六进制转译序列。

Rule 4.2　　禁止使用三字母词(trigraphs)

级别　　建议

分析　　可判定，单一编译单元

适用　　C90，C99

原理

三字母词(或叫三联符序列)由两个问号起始，后跟一个特定字符组成。截至目前(2020 年)，三字母词只有 9 个：

序号	三字母词	对应的字符
1	??=	#
2	??(	[
3	??)	]
4	??<	{
5	??>	}
6	??/	/
7	??!	|
8	??'	^
9	??-	~

源代码中的“三字母词”，在编译阶段会被替换为“对应的字符”。

而它们会与两个问号的其他用法引起意外混淆。

小贴士：*复合字母(“<:”　“:>”　“<%”　“%>”　“%:”　“%:%:”)是被允许的，因为它们只是标记符。*

示例

例如，字符串

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

会被编译器解析为

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

建议：三字母词几乎不被使用，但我们要注意避免可能的误用，就如上面的例子。

8.5 标识符(Identifiers)

Rule 5.1 外部标识符不得重名

C90 [Undefined 7] , C99 [Unspecified 7; Undefined 28]

级别 必要

分析 可判定，系统范围

适用 C90，C99

展开

本准则要求不同的外部标识符在实现的有效范围内的字符是不同的。

“不重名”取决于实现和所使用的 C 语言版本：

◆ 在 C90 中，最小有效字符范围是前 6 个字符，且不区分大小写；

◆ 在 C99 中，最小有效字符范围是前 31 个字符，而其通用字符和扩展字符的有效范围是 6 到 10 个字符。

实际上，许多实际的使用环境(编译环境)提供了更大的限制。 例如，通常 C90 中的外部标识符区分大小写，并且至少前 31 个字符有意义。

原理

如果两个标识符仅在非有效字符上不同，则实际行为无法确定。

如果考虑到可移植性，则应谨慎使用“标准”中指定的最低限制来应用此规则。

长标识符可能会损害代码的可读性。 尽管许多自动代码生成系统生成的标识符很长，但是有一个很好的论据可以将标识符长度保持在此限制之下。

小贴士：在 C99 中，如果扩展的源字符出现在外部标识符中并且该字符没有对应的通用字符，则标准不会指定它占用了多少个字符。

示例

在以下示例中，所有定义均出现在同一翻译单元中。 该实现中外部标识符中支持 31 个区分大小写的字符。

```
/*      12345678901234567890123456789012*********   Characters */
int32_t engine_exhaust_gas_temperature_raw;
int32_t engine_exhaust_gas_temperature_scaled;      /*  违 规 */
/*      12345678901234567890123456789012*********   Characters */
int32_t engine_exhaust_gas_temp_raw;
int32_t engine_exhaust_gas_temp_scaled;             /* 合 规 */
```

在以下违规示例中，该实现在外部标识符中支持 6 个不区分大小写的字符。 两个编译单元中的标识符

不同，但在主要特征上并没有区别。

```
/* file1.c */
int32_t abc = 0;

/* file2.c */
int32_t ABC = 0;
```

参阅

Dir 1.1， Rule 5.2， Rule 5.4， Rule 5.5

建议：全局变量、宏、全局函数等，均需符合此准则，以 C99 为例，前 31 个字符必须不相同，一个有效的办法是，命名少于 31 个字符，且不重名。

Rule 5.2 同范围和命名空间内的标识符不得重名

C90 [Undefined 7]，C99 [Undefined 28]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

如果两个标识符都是外部标识符，则本准则不适用，因为此情况适用于 Rule 5.1。

如果每个标识符都是宏标识符，则本准则不适用，因为这种情况已被 Rule 5.4 和 Rule 5.5 涵盖。

“不重名”的定义取决于实现和所使用的 C 语言版本：

- 在 C90 中，最低要求是前 31 个字符有效。
- 在 C99 中，最低要求是前 63 个字符有效，通用字符或扩展源字符视为一个字符。

原理

如果两个标识符仅非有效字符不同，则其实际行为不可预知。

考虑到出错的可能性，应谨慎使用“标准”中规定的最小限制来应用本准则。

长标识符可能会损害代码的可读性。 尽管许多自动代码生成系统生成的标识符很长，但是有一个很好的论据可以将标识符长度保持在此限制之下。

示例

在下面的示例中，所讨论的实现为：在不具有全局属性的标识符中支持 31 个区分大小写的字符。

```
/*            12345678901234567890123456789012********          Characters */
extern int32_t engine_exhaust_gas_temperature_raw;
static int32_t engine_exhaust_gas_temperature_scaled;               /* 违规 */
```

```
void f(void)
{
    /*      12345678901234567890123456789012********                Characters */
    int32_t engine_exhaust_gas_temperature_local;                    /* 合 规 */
}
/*             12345678901234567890123456789012********          Characters */
static int32_t engine_exhaust_gas_temp_raw;
static int32_t engine_exhaust_gas_temp_scaled;                       /* 合 规 */
```

参阅

Dir 1.1, Rule 5.1, Rule 5.3, Rule 5.4, Rule 5.5

建议：为简化编写规则，可以采用与 Rule 5.1 相同的方案，限制命名长度为 31 个字符。

Rule 5.3　内部声明的标识符不得隐藏外部声明的标识符

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

内部范围声明的标识符应与外部范围声明的标识符不重名。

“不重名”的定义取决于实现和所使用的 C 语言版本：

- 在 C90 中，最低要求是前 31 个字符有效。
- 在 C99 中，最低要求是前 63 个字符有效，通用字符或扩展源字符视为一个字符。

原理

如果在内部作用域中声明一个标识符，与在外部作用域中已经存在的标识符重名，则最内部的声明将“隐藏”外部的声明。 这可能会导致开发人员混乱。

小贴士：一个命名空间中声明的标识符不会隐藏在另一命名空间中声明的标识符。

术语“内部作用域”和“外部作用域”定义如下：

- 文件间识别的范围，作用域最大，亦即最大范围的外部作用域；
- 文件内模块间可识别的范围，作用域较小，相对最大范围，视为内部作用域；
- 仅在模块内识别的范围，作用域最小，上述两种情况相对这种情况都为外部作用域。

示例

```
void fn1 (void)
{
    int16_t i;                  /* 定义变量 "i" */
```

```
    {
        int16_t i;              /* 违规 - 会隐藏前面定义的 "i" */
        i = 3;                  /* 这里的 "i" 是指前面哪一个，会产生混淆 */
    }
}
struct astruct
{
    int16_t m;
};
extern void g(struct astruct *p);
int16_t xyz = 0;                /* 定义变量 "xyz" */
void fn2 (struct astruct xyz)   /* 违规 - 外部定义的 "xyz" 被同名形参隐藏 */
{
    g(&xyz);
}
uint16_t speed;
void fn3(void)
{
    typedef float32_t speed;    /* 违规 - 类型将变量给隐藏 */
}
```

参阅

Rule 5.2, Rule 5.8

Rule 5.4　宏标识符不得重名

C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

本准则要求在定义一个宏时，其命名必须不同于

- 已定义的其他宏的名称，和
- 已定义的参数的名称。

它还要求给定宏的参数名称彼此不同，但不要求宏参数名称在两个不同的宏之间不同。

“不重名”的定义取决于实现和所使用的 C 语言版本：

- 在 C90 中，最低要求是宏标识符的前 31 个字符有效。
- 在 C99 中，最低要求是宏标识符的前 63 个字符有效。

实际上，在实际实现时可以设置更大的限制。本准则要求宏标识符在实现所施加的限制内是唯一的。

原理

如果两个宏标识符仅非有效字符不同，则其实际实现行为不可预知。 由于宏参数仅在其宏扩展期间处于活动状态，因此一个宏中的参数与另一宏中的参数混淆不会有问题。

考虑到可移植性，则应谨慎使用“标准”中指定的最低限制来应用此准则。

过长的宏标识符可能会损害代码的可读性。 尽管许多自动代码生成系统会生成很长的宏标识符，但有一个很好的论据认为宏标识符的长度应远低于此限制。

小贴士：在 C99 中，如果扩展源字符出现在宏名称中，并且该字符没有对应的通用字符，则标准不会指定它占用多少个字符。

示例

在以下示例中，讨论的实现为：宏标识符中支持 31 个区分大小写的有效字符。

```
/*      12345678901234567890123456789012********    Characters */
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s  /* 违规   */
/*      12345678901234567890123456789012********    Characters */
#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* 合 规 */
```

参阅

Rule 5.1 , Rule 5.2, Rule 5.5

建议：与 Rule5.1,5.2,5.3 相同，限制宏命名长度为 31 个字符

Rule 5.5　宏标识符与其他标识符不得重名

C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

本准则要求在预编译前存在的宏标识符与预编译后的标识符也不能相同。它适用于任何标识符(不限作用域与命名空间)和已定义的任何宏，不论声明该标识符时该定义是否仍然有效。宏标识符的生命周期为预编译期间，预编译结束后，执行编译时，宏已经失效。即便如此，仍然不允许有标识符与已经失效的宏重名。

根据所使用的C 语言的实现和版本，“不重名”的定义不相同：

◆ 在 C90 中，最低要求是前 31 个字符有效。

◆ 在 C99 中，最低要求是前 63 个字符必须有效，每个通用字符或扩展源字符都视为一个字符。

原理

宏名称和标识符保持不同有助于避免开发人员混淆。

示例

在下面的违规示例中，类似函数的宏 Sum 的名称也用作标识符。对象(变量)Sum 的声明不进行宏展开，因为它后面没有“(”字符。因此，标识符在进行预处理后仍存在。

```
#define Sum(x, y) ((x) + (y))
int16_t Sum;
```

以下示例合规，因为在预处理后将不再存在名为 Sum 的实例。

```
#define Sum(x, y) ((x) + (y))
int16_t x = Sum (1, 2);
```

在下面的示例中，假定有效字符长度为 31 且区分大小写。该示例违规，因为宏名称与标识符的前 31 个字符完全相同。

```
/*            12345678901234567890123456789012******** Characters */
#define       low_pressure_turbine_temperature_1 lp_tb_temp_1
static int32_t low_pressure_turbine_temperature_2;
```

参阅

Rule 5.1, Rule 5.2, Rule 5.4

Rule 5.6　typedef 名称应是唯一标识符

级别　必要

分析　可判定，系统范围

适用　C90，C99

展开

一个 typedef 名称在所有命名空间和编译单元中都应该是唯一命名的。仅当 typedef 名称定义在头文件中，而该头文件被多个源文件包含时，才允许出现多个声明使用同一个 typedef 名称。这种情况，这多个声明的类型是一致的，即不存在多个类型命名为同一个名称的情况。而且头文件会通过预防措施，防止其被多次包含(见 Dir 4.10)，所以定义在头文件中的 typedef 名称，其实也是唯一声明的。

原理

如果多个 typedef 名称命名相同而它们实际指代又是不同的函数、对象或枚举常量时，开发人员会被困扰。

例外

typedef 名称可以与和 typedef 相关联的结构、联合或枚举标记(tag)名称相同。

示例

```
void func ( void )
{
    {
        typedef unsigned char u8_t;
    }
    {
        typedef unsigned char u8_t; /* 违规 - 重复使用 */
    }
}
typedef float mass;
void func1 ( void )
{
    float32 t mass = 0.0f;          /* 违规 - 重复使用 */
}
typedef struct list
{
    struct list *next;
    uint16_t element;
} list;                             /* 合规 - 符合例外的情况 */
typedef struct
{
    struct chain
    {
        struct chain *list;
        uint16_t element;
    } s1;
    uint16_t length;
} chain;                            /* 违规 - 标记 "chain" 与 typedef 不关联 */
```

参 阅

Rule 5.7

Rule 5.7　标签(tag)名称应是唯一标识符

级别　必要

分析　可判定，系统范围

适用　C90，C99

展开

标签(tag)在所有名称空间和单位编译中应是唯一的。

标签(tag)的所有声明应指定相同的类型。

仅当在头文件中声明标签(tag)并且该头文件被多个源文件包含时，此准则才允许使用同一标签(tag)的多个完整声明。与 typedef 名称相同，这种情况下，所有声明都是完全相同的，且由于头文件防止多次包含的举措，也不会出现多个声明。

原理

重用标签(tag)名称可能会导致开发人员混乱。

尽管在标准附件中未列出，但在 C90 中还存在与标签名称重用相关的不确定行为。这种未定义的行为在 C99 第 6.7.2.3 节中被定义为约束条件。

例外

标签(tag)名称可能与与其关联的 typedef 名称相同。与 Rule 5.6 相同。

示例

```
struct stag
{
    uint16_t a;
    uint16_t b;
};
struct stag a1 = { 0, 0 };  /* 合规 - 与前面的定义一致 */
union stag a2 = { 0, 0 };   /* 违规 - 与声明的 struct stag 不一致。
                             * 同时也违背了C99的约束 */
```

下述的示例也违背了 Rule 5.3：

```
struct deer
{
    uint16_t a;
    uint16_t b;
};
void foo ( void )
{
    struct deer
    {
        uint16_t a;
    };                  /* 违规 - 标签 "deer" 重复使用 */
}
typedef struct coord
{
    uint16_t x;
    uint16_t y;
} coord;                /* 合规 - 符合例外情况 */
```

```
struct elk
{
    uint16_t x;
};
struct elk              /* 违规 - 同一个标签被声明为不同的类型
                         * 同时也违背了C99的约束 */
{
    uint32_t x;
};
```

参 阅

Rule 5.6

Rule 5.8 定义具有外部链接的对象或功能的标识符应是唯一的

级别 必要

分析 可判定，系统范围

适用 C90，C99

展开

用作外部标识符的标识符不得在任何命名空间或编译单元中用于任何其他目的，即使它没有链接的对象。

原理

强制标识符名称的唯一性有助于避免混淆。没有链接对象的标识符不必是唯一的，因为混淆风险很小。**示例**

下面的示例中，“file1.c”和“file2.c”是同一个项目的一部分。

```
/* file1.c */
int32_t count;                  /* "count" 具有全局属性(全局变量)          */
void foo ( void )               /* "foo" 具有全局属性(全局函数)            */
{
    int16_t index;              /* "index" 无全局属性(临时变量)            */
}


/* file2.c */
static void foo ( void )        /* 违规 - “ foo”不唯一(在file1.c中有全局属
                                 * 性的同名函数) */
{
    int16_t count;              /* 违规 - "count" 没有全局属性，但与另一文
                                 * 件的有全局属性的变量重名 */
    int32_t index;              /* 合规 - "index"无全局属性(临时变量)      */
```

```
}
```

补充：外部链接(internal linkage)翻译为全局，不被“static”修饰的全局变量和函数。

参阅

Rule 5.3

Rule 5.9 定义具有内部链接的对象或函数的标识符应是唯一的

级别 建议

分析 可判定，系统范围

适用 C90，C99

展开

标识符名称在所有命名空间和编译单元中都应该唯一。任何标识符都不应与任何其他标识符具有相同的名称，即使该其他标识符没有链接的对象也是如此。

原理

强制标识符名称的唯一性有助于避免混淆。

例外

可以在一个以上的转换单元中定义具有局部属性的内联函数，条件是所有这些定义都在每个转换单元中包含的同一头文件中进行。即：定义在头文件中的内联函数，不受限制。

示例

下面的示例中，“file1.c”和“file2.c”是同一个项目的一部分。

```
/* file1.c */
static int32_t count;        /* "count" 局部全局属性 */
static void foo ( void )     /* "foo" 局部全局属性 */
{
    int16_t count;           /* 违规 - "count" 没有全局属性，但与有局部全
                              * 局属性的标识符冲突 */
    int16_t index;           /* "index" 无全局属性 */
}
void bar1 ( void )
{
    static int16_t count;    /* 违规 - "count" 没有全局属性，但与有局部全
                              * 局属性的标识符冲突 */
    int16_t index;           /* 合规 - "index" 不唯一但它没有与其冲突的具全
                              * 局属性的标识符 */
    foo ( );
}
```

```
/* End of file1.c */

/* file2.c */
static int8_t count;        /* 违规 - "count" 具有局部全局属性，与另一个
                             * 具有局部全局属性的标识符重复 */
static void foo ( void )    /* 违规 - "foo" 具有局部属性，与另一个具有局
                             * 部属性的函数标识符重复 */
{
    int32_t index;          /* 合规 - "index" 和 "nbytes" */
    int16_t nbytes;         /* 不唯一，但因都不具全局属性，因而不冲突 */
}
void bar2 ( void )
{
    static uint8_t nbytes;  /* 合规 - "nbytes" 不唯一，但它没有全局属性，
                             * 全局属性与存储类别无关 */
}
/* End of file2.c */
```

补充：内部链接(internal linkage)翻译为局部全局，即被“static”修饰的全局变量和函数，不包括函数内部定义的静态变量。

参 阅

Rule 8.10

8.6 类型(Types)

Rule 6.1 位字段只能用适当的类型声明

C90 [Undefined 38; Implementation 29], C99 [Implem entation J.3.9(1, 2)]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

“适当的”位域类型为：

◆ C90：unsigned int 或 signed int；

◆ C99：下列几种之一：

- unsigned int 或 signed int；
- 实现允许的其他显示声明的有符号或无符号整数类型；

- _Bool(C99 新增的布尔类型“bool”是它的同名宏，详见 C99 stdbool.h)；

小贴士：允许使用 typedef 来定义适当的类型。

原理

“int”随具体实现(编译环境等)的不同可能为有符号也可能为无符号。

C90 中不允许使用 enum、short、char 或其他任何类型，是因为它们的具体实现无法确定。不同的编译环境，对这些类型的符号和数据长度定义会不一致。

C99 中，实现可以定义位域声明中允许的其他整数类型。一般的 C99 编译环境都会有选项，供选择 enum、short、char 是否有符号和数据长度，这些确认后，其是否可用于位域声明自然也就明确了。

注意：为了兼容性考虑，即使是 C99，也不建议使用需要依赖具体实现(编译环境)来定义的类型。

示例

以下示例适用于不提供任何其他位域类型的 C90 和 C99 实现。假定 int 类型为 16 位。

```
typedef unsigned int UINT_16;
struct s {
    unsigned int b1:2; /* 合规                                    */
    int          b2:2; /* 违规 - 不允许使用不明确符号的"int"      */
    UINT_16      b3:2; /* 合规 - 由typedef声明的"unsigned int"    */
    signed long  b4:2; /* 违规 - 即使 long 和 int 大小相同        */
};
```

Rule 6.2	单比特(single-bit)位域成员不可声明为有符号类型

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

根据 C99 标准第 6.2.6.2 节，一个单比特(single-bit)带符号的位域数据具有 1 个符号位和 0 个值位。在任何整数表示中，0 个值位都无法指定有意义的值。

因此，一个单比特(single-bit)带符号的位域数据不太可能以有用的方式允许，而且它的存在也会给程序员带来困惑。

虽然 C90 标准没有提供太多有关类型表示的详细信息，但与 C99 相同的注意事项也适用。

小贴士：本准则不适用于未命名的位域数据，因为它们无法访问。

8.7 字符和常量(Literals and constants)

Rule 7.1　禁止使用八进制常数

[Koenig 9]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

开发人员编写的常量前导零可能会被误读为十进制常量。八进制常数的表述方式：0**。

小贴士：此准则不适用于八进制转义序列，因为使用前导“\”字符混淆的范围较小。 **例外**

整数常数零(写为“0”)严格来说是八进制常数，但是是本准则的允许例外。

示例

```
extern uint16_t code[10];
code[1] = 109; /* 合 规 - 十 进 制 数 109 */
code[2] = 100; /* 合 规 - 十 进 制 数 100 */
code[3] = 052; /* 违 规 - 十 进 制 数 42 */
code[4] = 071; /* 违 规 - 十 进 制 数 57 */
```

Rule 7.2　后缀“u”或“U”应使用于所有无符号的整数常量

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

本规则适用于：

- 出现在#if 和#elif 预处理指令的控制表达式中的整数常量；
- 预处理后存在的任何其他整数常量。

小贴士：在预处理期间，整数常量的类型与预处理之后的确定方式相同，除了：

- *所有带符号的整数类型的假定为long(C90)或 intmax_t(C99)；*
- *所有无符号整数类型的假定为 unsigned long(C90)或 uintmax_t(C99)。*

原理

整数常量的类型是潜在的混淆源，因为它取决于多种因素的复杂结合，包括：

- 常数的大小；
- 整数类型的实现大小；
- 任何后缀；
- 表示值的数字基数(即十进制，八进制或十六进制)。

小贴士：

- *任何带有“U”后缀的值均为无符号类型；*
- *小于 2^{31} 的无后缀十进制值为带符号类型。*

但：

- *大于或等于 2^{15} 的无后缀十六进制值可以是有符号或无符号类型；*
- *对于 C90，大于或等于 2^{31} 的无后缀十进制值可能是带符号或无符号类型。*

常量的符号应该是明确的。如果常量是无符号的，则加上“U”后缀可以使程序员清除的知道该常量无符号。

小贴士：此规则不受使用常量的上下文影响；(计算时的值域)提升和可能应用于该常数的其他转换与确定是否遵守此规则无关。

示例

以下示例假定计算机具有 16 位 int 类型和 32 位 long 类型。它显示了根据标准确定的每个整数常量的类型。整数常量 0x8000 不符合标准，因为它具有无符号类型，但没有“ U”后缀。

常数	类型	是否合规
32767	signed int	合规
0x7fff	signed int	合规
32768	signed long	合规
32768u	unsigned int	合规
0x8000	unsigned int	违规
0x8000u	unsigned int	合规

Rule 7.3 小写字符“l”不得作为常量的后缀使用(仅可使用“L”)

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

在声明文字时，使用大写后缀“L”可消除“1”(数字 1)和“l”(字母“ el”)之间的潜在歧义。

示例

小贴士：示例中的 long long 后缀仅适用于C99。

```
const int64_t    a = 0L;
const int64_t    b = 0l;    /* 违规 */
const uint64_t   c = 0Lu;
const uint64_t   d = 0lU;   /* 违规 */
const uint64_t   e = 0ULL;
const uint64_t   f = 0Ull;  /* 违规 */
const int128_t   g = 0LL;
const int128_t   h = 0ll;   /* 违规 */
const float128_t m = 1.2L;
const float128_t n = 2.4l;  /* 违规 */
```

Rule 7.4 除非对象的类型为“指向 const char 的指针”，否则不得将字符串常量赋值给该对象

C90 [Undefined 12], C99 [Unspecified 14; Undefined 30]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

不得尝试直接修改字符串常量或宽字符串常量。

除非对象的类型为“指向 const char 型数组的指针”，否则对字符串常量取地址(&运算符修饰)的结果不得赋值给该对象。

同样的，此限制也适用于宽字符串常量。 除非对象的类型是“指向 const wchar_t 的指针”，否则不得将宽字符串常量赋值给该对象。 除非该对象的类型是“指向 const wchar_t 数组的指针”，否则对宽字符串常量取地址(&运算符)的结果不得赋值给该对象。

原理

对字符串常量的修改结果无法预知。例如，某些实现可能会将字符串常量存储在只读存储器中，在这

种情况下，尝试修改字符串常量将失败，并且还可能导致异常或崩溃。

此规则可防止误修改字符串常量。

在 C99 标准中，并未明确指定是否将共享共同结尾的字符串常量存储在不同的内存位置中。因此，即使尝试修改字符串常量看起来成功了，也可能会无意中更改了另一个字符串。

示例

下面示例显示了直接修改字符串常量的尝试：

```
"0123456789"[0] = '*';       /* 违规 */
```

这些示例描述了如何防止间接修改字符串常量。

```
/* 违规 - s 缺少 const 修饰 */
char *s = "string";
/* 合规 - p 有 const 修饰；其他限定词是可接受的 */
const volatile char *p = "string";

extern void f1(char *s1);
extern void f2(const char *s2);
void g(void)
{
    f1("string");           /* 违规 - 形参 s1 缺少 const 修饰     */
    f2("string");           /* 合规                               */
}
char *name1(void)
{
    return ("MISRA");       /* 违规 - 返回类型缺少 const 修饰     */
}
const char *name2(void)
{
    return ("MISRA");       /* 合规                               */
}
```

参阅

Rule 11.4, Rule 1 1.8

8.8 声明和定义(Declarations and definitions)

Rule 8.1 类型须明确声明

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

C90 标准允许在某些情况下省略类型，在这种情况下，将隐式指定 int 类型。 可能使用隐式 int 的情况示例如下：

- 对象声明；
- 形参声明；
- 成员声明；
- typedef 声明；
- 函数返回类型。

省略显式类型声明可能会导致混淆。 例如，在声明中：

```
extern void g(char c, const k);
```

k 的类型隐式定义为 const int，而实际可能期望为 const char。

示例

以下示例显示了合规和违规的对象声明：

```
extern         x; /* 违规 - 隐式int类型   */
extern int16_t x; /* 合规 - 显式类型      */
const          y; /* 违规 - 隐式int类型   */
const int16_t  y; /* 合规 - 显式类型      */
```

以下示例显示了合规和违规的函数类型声明：

```
extern f(void);                         /* 违规 - 返回类型隐式声明为int型 */
extern int16_t f(void);                 /* 合规                           */
extern void g(char c, const k);         /* 违规 - 形参k隐式声明为int型    */
extern void g(char c, const int16_t k); /* 合规                           */
```

以下示例显示了合规和违规的类型定义：

```
typedef (*pfi)(void);           /* 违规 - 返回类型隐式声明为int          */
typedef int16_t (*pfi)(void);   /* 合规                                  */
typedef void (*pfv)(const x);   /* 违规 - 形参x隐式声明为int             */
typedef void (*pfv)(int16_t x); /* 合规                                  */
```

以下示例显示了合规和违规的成员声明：

```
struct str
{
    int16_t x;  /* 合规                          */
    const   y;  /* 违规 - 成员y隐式声明为int  */
} s;
```

参 阅

Rule 8.2

Rule 8.2　函数类型应为带有命名形参的原型形式

C90 [Undefined 22－25], C99 [Undefined 36－39, 73, 79]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

早期的 C 版本，通常称为K&R C [30]，没有提供一种机制来检查形参的数量或类型。对象或函数的类型不必在 K&R C 中声明，因为对象的默认类型和函数的默认返回类型都为 int。

C90 标准引入了函数原型，这是一种声明了形参类型的函数声明形式。这使得 C90 允许(编译环境)根据声明的形参类型进行形参类型检查。同样的，除非函数原型指定可变数量的形参，它也允许检查形参数量。由于代码向后兼容的需要，C90 可以不使用函数原型。出于相同的原因，它也允许省略类型，在这种情况下， 类型将隐式声明为 int。

C99 标准移除了默认的 int 形参，但它通过可设置的选项提供了对旧式的 K&R 样式的函数的支持。

实参和形参的数量、类型和函数的预期返回类型、实际返回类型之间的不匹配，为潜在的未定义行为提供了可能。本规则与 Rule 8.1 和 Rule 8.4 的目的是通过要求明确指定形参类型和函数返回类型来避免这种未定义行为。Rule17.3 则保证了在调用函数时可以获取这些信息，从而可以一起要求编译器诊断出任何的不匹配情况。

本规则还要求为声明中的所有形参指定名称。形参名称可以提供相关功能接口的有用信息，声明与定义间的不匹配可能预示着编程错误。

小贴士：空的参数列表在原型中无效。如果一个函数类型没有参数，则应在函数原型中使用关键字“void”来显示声明。

示例

第一个示例显示了一些函数的声明以及其中一些函数的相应定义。

```
/* 合规                                                    */
extern int16_t func1(int16_t n);
/* 违规 - 未指定形参名称                                   */
extern void func2(int16_t);
/* 违规 - 不是有效的原型形式(空形参须明确填入"void")       */
static int16_t func3 ();
/* 合规 - 原型明确指定了无形参                             */
static int16_t func4(void);
/* 合规                                                    */
int16_t func1(int16_t n)
{
    return n;
}
/* 违规 - 旧式的标识符和声明列表                           */
static int16_t func3(vec, n)
int16_t *vec;
int16_t n;
{
    return vec[n - 1];
}
```

下面的示例描述了此规则在函数类型和函数声明和定义之外的应用：

```
/* 违规 - 无效的原型形式(空形参须明确填入"void")          */
int16_t (*pf1)();
/* 合规 - 原型明确指定了无形参                             */
int16_t (*pf1)(void);
/* 违规 - 未指定形参名称                                   */
typedef int16_t (*pf2_t)(int16_t);
/* 合规                                                    */
typedef int16_t (*pf3_t)(int16_t n);
```

参阅

Rule 8.1, Rule 8.4, Rule 17.3

Rule 8.3　对象或函数的所有声明均应使用相同的名称和类型限定符

C90 [Undefined 10] , C99 [Undefined 14], [Koenig 59－62]

级别　必要

分析　可判定，系统范围

适用　C90，C99

展开

存储类说明符不在此规则的限定范围内。

原理

同一对象或函数的声明使用一致类型和限定符会使代码更强壮。

在函数原型中指定形参名称可以检查函数定义及其声明的接口一致性。

例外

相同基本类型的兼容版本可以互换使用。 例如，int，signed 和 signed int 都是等效的。但仍建议显式使用完全相同的描述。

示例

```
extern void f(signed int);
void f(int);                    /* 合规 - 符合例外情况                */
extern void g(int * const);
void g(int *);                  /* 违规 - 类型限定词不一致            */
```

小贴士：上面的这些示例都违背了Dir 4.6。

```
extern int16_t func(int16_t num, int16_t den);
/* 违规 - 形参命名不一致                        */
int16_t func(int16_t den, int16_t num)
{
    return num / den;
}
```

在下面示例中，定义与声明中使用的形参 h 类型名不同。 即使 width_t 和 height_t 是相同的基本类型，也不符合规则。

```
typedef uint16_t width_t;
typedef uint16_t height_t;
typedef uint32_t area_t;
extern area_t area(width_t w, height_t h);
area_t area(width_t w, width_t h)   /* 违规 - 形参的类型不一致 */
{
    return (area_t)w * h;
}
```

此规则不要求函数指针声明使用与函数声明相同的名称。 因此，以下示例是合规的。

```
extern void f1(int16_t x);
extern void f2(int16_t y);
void f(bool_t b)
{
    void (*fp1)(int16_t z) = b ? f1 : f2;
}
```

参 阅

Rule 8.4

Rule 8.4 全局(external linkage)的对象和函数，应有显式的合规的声明

C90 [Undefined 24], C99 [Undefined 39]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

兼容声明是为要定义的对象或函数描述其兼容类型的声明。

原理

如果在定义对象或函数时其声明已可见，则编译器必须检查该声明和定义是否合规。根据规则 8.2 的要求，在功能原型存在的情况下，检查应扩展到形参的数量和类型。

给全局对象和函数添加声明的建议方法是：在头文件中声明它们，然后将头文件包含在所有需要它们的代码文件中，包括定义它们的代码文件(参见 Rule 8.5)。

示例

下面这些示例中，除了代码中存在的对象或函数外，没有其他对象或函数的声明或定义。

```
extern int16_t count;
       int16_t count = 0;        /* 合规                                    */
extern uint16_t speed = 6000u;   /* 违规 - 定义前未声明                     */
uint8_t pressure = 101u;         /* 违规 - 定义前未声明                     */
extern void func1(void);
extern void func2(int16_t x, int16_t y);
extern void func3(int16_t x, int16_t y);
void func1(void)
{
    /* 合规 */
}
void func2(int16_t x, int16_t y)
{
    /* 合规 */
}
void func3(int16_t x, uint16_t y)
{
    /* 违规 - 形参类型不一致，违反了 Rule 8.3，即虽有声明但声明违规       */
}
void func4(void)
{
    /* 违规 - 定义前未声明                                                */
}
```

```
static void func5(void)
{
    /* 合规 - 此规则不适用于局部全局(internal linkage)的对象和函数          */
}
```

参阅

Rule 8.2, Rule 8.3, Rule 8.5, Rule 17.3

Rule 8.5　全局对象或函数应在且只在一个文件中声明一次

[Koenig 66]

级别　必要

分析　可判定，系统范围

适用　C90，C99

展开

此规则仅适用于非定义声明。

原理

将单一的声明放在头文件中，该头文件再被任何需要调用该标识符的编译单元包含，这是种通常做法。这种做法可以确保：

- 声明和定义的一致性；
- 不同编译单元中的声明的一致性。

小贴士：一个项目中可能会有很多的头文件，但是每个外部对象和函数只能在一个头文件中进行声明。

示例

```
/* featureX.h */
extern int16_t a;    /* 声明 a */

/* file.c  */
#include "featureX.h"
int16_t a = 0;       /* 定义 a */
```

参 阅

Rule 8.4

Rule 8.6　全局标识符应在且只在一处定义

C90 [Undefined 44], C99 [Undefined 78], [Koenig 55, 63－65]

级别　必要

分析　可判定，系统范围

适用　C90，C99

原理

存在多个定义的标识符或是没有定义的标识符，其实现行为都是未定义的。即使定义相同，此规则也不允许在不同文件中使用多个定义(编译器也不允许)。如果有多个声明且声明有不同，则其行为也为未定义。

示例

下面的示例中，对象“i”被定义了两次。

```
/* file1.c */
int16_t i = 10;
/* file2.c */
int16_t i = 20;      /* 违规 - 全局变量 i 有两处定义 */
```

下面的示例中，对象“j”有一个临时定义和一个全局定义。

```
/* file3.c */
int16_t j;           /* Tentative definition            */
int16_t j = 1;       /* Compliant - external definition */
```

建议：此 file3.c 的示例，并非一般定义方式，建议不要使用这种用法。

以下示例是违规的，因为对象 k 有两个外部定义。 file4.c 中的临时定义在翻译单元的末尾成为外部定义。

```
/* file4.c */
int16_t k;           /* Tentative definition - becomes external  */
/* file5.c */
int16_t k = 0;       /* External definition                      */
```

建议：此示例，并非一般定义方式，建议不考虑(Tentative definition)，而采用一般的定义，然后使用它的做法。

Rule 8.7　仅在本编译单元中调用的对象和函数，应定义成局部属性

[Koenig 56, 57]

级别　建议

分析　可判定，系统范围

适用　C90，C99

原理

将对象定义为局部全局(internal linkage)或临时(no linkage)属性，可以限制对象的可见范围，从

而减少无意中访问该对象的机会。同理，赋予函数局部属性，也可降低其可见性，从而减少无意中调用该函数的机会。

遵守此规则还避免了不同翻译单元或库的同名标识符之间出现混淆的可能。

Rule 8.8　“static”修饰符应用在所有局部全局对象和局部函数(internal linkage)的声明中

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则同时适用于声明和定义。

原理

C 语言标准指出，如果用外部类说明符声明了一个对象或函数，并且该对象或函数的另一个声明已经可见，则链接时会由较早的声明指定。这可能会造成混淆，因为一般情况下我们会期望用外部存储类说明“extern”符创建外部链接。因此，静态存储类说明符“static”应始终应用于局部全局对象和局部函数(具有内部链接的对象和功能)。加“static”修饰的一个好处是，我们可以防止外部错误的访问或调用局部全局变量和局部函数。尤其是这些变量和参数在我们的设计时有严格的访问、调用时序限制时。

示例

```
static int32_t x = 0;       /* 定义: 局部全局变量 internal linkage    */
extern int32_t x;           /* 违规                                   */
static int32_t f(void);     /* 声明: 局部函数 internal linkage        */
int32_t f(void)             /* 违规                                   */
{
    return 1;
}
static int32_t g(void);     /* 声明: 局部函数 internal linkage        */
extern int32_t g(void)      /* 违规                                   */
{
    return 1;
}
```

Rule 8.9　若一个对象的标识符仅在一个函数中出现，则应将它定义在块范围内

级别　建议

分析　可判定，系统范围

适用　C90，C99

原理

在块范围内定义对象可减少无意间访问该对象的可能性，并明确表示出“不应在其他位置访问该对象”的意图。

在一个函数中，对象是定义在最外层还是最内层很大程度上取决于个人风格。

人们认识到，在某些情况下可能无法遵守本规则。例如，在块范围内声明的具有静态存储属性的对象不能直接从块的外部访问。如果不使用对对象的间接访问，就不可能设置和检查单元测试用例的结果。在这种情况下，某些项目可能更倾向于不应用此规则。

示例

在下面的合规示例中，i 被声明在块范围内，因为它是循环计数。相同文件中的其他函数无法将其用作任何其他目的。

```
void func(void)
{
    int32_t i;
    for (i = 0; i < N; ++i)
    {
    }
}
```

在下面的合规示例中，函数计数跟踪被调用的次数并返回该数字。 其他函数不需要知道 count 实现的详细信息，因此调用计数器是使用块作用域定义的。

```
uint32_t count(void)
{
    static uint32_t call_count = 0;
    ++call_count;
    return call_count;
}
```

Rule 8.10　内联函数应使用静态存储类声明

级别　必要

分析　可判定，单一编译单元

适用　C99

原理

如果一个内联函数是使用外部链接声明的，但在不同编译单元中都有定义，则该行为是未定义的。 调用通过外部链接声明的内联函数可以调用函数的外部定义，也可以使用内联定义。 尽管这不应影响所调用函数的行为，但可能会影响执行时间，因此对实时程序产生影响。

小贴士：将内联函数的定义放在标头文件中，则该内联函数可被用于多个编译单元。

参 阅

Rule 5.9

Rule 8.11 声明具有外部链接的数组时，应明确指定其大小

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

展开

此规则仅适用于声明，不适用于定义。 定义数组的同时，通过初始化来隐式的指定其大小是被允许的。**原理**

尽管标准 C 允许使用不完整类型声明数组并访问其元素，但显式的确定数组大小更安全。为每个声明提供其大小信息，可以检查它们的一致性。它还可以允许静态检查器在无需分析多个编译单元的情况下执行某些数组边界分析。

示例

```
extern int32_t array1[10];      /* 合规          */
extern int32_t array2[];        /* 违规          */
```

Rule 8.12 在枚举列表中，隐式指定的枚举常量的值应唯一

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

隐式指定的枚举常量的值比其前一个值大 1。如果第一个枚举常量也是隐式指定的，则其值为 0。

显式指定的枚举常量的值为其指定的常量表达式的值。如果枚举列表中混合使用隐式和显式指定常量，则可能会出现重复值。这种重复值可能无意义，并可能导致意外的行为。

本规则要求明确枚举常量的重复值，从而明确其意图。

示例

在下面示例中，枚举常量 yellow 和 green 的值相同。

```
/* 违规 - 隐式的指定 yellow 与 green 的值相同                    */
enum colour { red = 3, blue, green, yellow = 5 };
```

```
/* 合规 - yellow 与 green 以显式的方式指定其值相同                    */
enum colour { red = 3, blue, green = 5, yellow = 5 };
```

Rule 8.13　指针应尽可能指向 const 限定类型

级别　建议

分析　不可判定，系统范围

适用　C90，C99

展开

除下述两种情况外，都应将指针限定为指向 const 限定类型：

- 指针用于修改被指向的对象；
- 通过以下两种方式之一将一个指针复制到另一个指针：
  - 内存分配；
  - 内存移动或复制函数。

为简单起见，此规则是根据指针及其指向的类型编写的。但是，它同样适用于数组及其包含的元素类型。除下述情况外，数组应有 const 限定类型的元素：

- 数组的任一元素会被修改；
- 数组会被复制给符合上述例外情况，指向非 const 限定类型的指针。

原理

此规则是确保不会无意间使用指针来修改对象的最佳推荐。从概念讲，它等效于声明了：

- 所有数组必须具有 const 限定类型的元素，和
- 所有指针须指向 const 限定类型

然后，仅在有必要遵守语言标准的约束时，才移除 const 限定。

理解：我们在声明指针和数组时，除上述的例外情况外，应声明为 const 限定类型，而后，在实际实现时，才移除 const 限定以进行修改操作。

示例

下面的违规示例中，指针p 不用于修改对象，但它指向的类型没有 const 限定。

```
uint16_t f(uint16_t *p)
{
    return *p;
}
```

此函数以下面方式定义，那它就是合规的：

```
uint16_t f(const uint16_t *p)
```

下面的示例则违背了C 语言的限制，它使用指向 const 限定类型的指针来修改对象。

```
void h(const uint16_t *p)
{
    *p = 0;
}
```

下面的示例中，指针本身是 const 限定的，但其指向的类型不是，而指针 s 不用于修改对象，因此违规。

```
#include <string.h>
char last_char(char * const s)
{
    return s[strlen(s) - 1u];
}
```

若函数以下面方式定义，则此段代码合规：

```
char last_char(const char * const s)
```

在下面违规示例中，数组a 的所有元素均未被修改，但元素类型没有const 限定。

```
uint16_t first(uint16_t a[5])
{
    return a[0];
}
```

将函数以下面方式定义后，此段代码合规：

```
uint16_t first(const uint16_t a[5])
```

补充：数组做形参在编译时会弱化为指针，方括号内的数值不会被检查，故实际上上面的数组形参 a 直接定义成指针后，效果完全一样。若希望防止访问数组越界，则可以定义为下面的样式：

```
uint16_t first(const uint16_t *a, uint16_t length);或者
uint16_t first(const uint16_t a[], uint16_t length)
```

Rule 8.14　不得使用类型限定符“restrict”

C99 [Undefined 65, 66]

级别　必要

分析　可判定，单一编译单元

适用　C99

原理

谨慎的使用“restrict”类型限定符可以提高编译器编译代码的效率。它还可以优化静态分析。但是，要使用“restrict”类型限定符，程序员必须确保用两个或多个指针操作的内存区域不会重叠。

如果未正确使用“restrict”，则会存在很大的编译器生成与预期不符的代码的风险。

示例

MISRA C 准则不适用于标准库函数，故下面示例合规。但是，程序员必须确保由 p、q、和 n 定义的区域不重叠。

```
#include <string.h>
void f(void)
{
    /* memcpy 具有“restrict”类型的形参 */
    memcpy(p, q, n);
}
```

下面的示例违规，它使用了“restrict”类型限定符。

```
void user_copy(void * restrict p, void * restrict q, size_t n)
{
}
```

8.9 初始化(Initialization)

Rule 9.1 具有自动存储持续时间的对象(临时变量)的值在设置前不得读取

C90 [Undefined 41], C99 [Undefined 10, 17]

级别 强制

分析 不可判定，系统范围

适用 C90，C99

展开

在本规则中，数组元素或结构体成员应被视为离散对象。

原理

根据 C 标准，具有静态存储持续时间(静态变量)在非明确初始化时将自动初始化为零；具有自动存储持续时间(临时变量)则不会自动初始化，因此可能具有不确定的值。

小贴士：在某些情况下，有可能发生忽略自动对象(临时变量)的显示初始化的情况。比如，当使用 goto 或 switch 语句跳转到标签时，就有可能“绕过”对象的声明；这种情况下，对象的声明被认为是符合预期的(即不会报编译错误)，但是任何显式的初始化都将被忽略。

示例

```
void f(bool_t b, uint16_t *p)
{
    if(b)
```

```
    {
        *p = 3U;
    }
}
void g(void)
{
    uint16_t u;
    f(false, &u);
    if(u == 3U)
    {
        /* 违规 - u 尚未赋值              */
    }
}
```

在下面的不符合 C99 的示例中，goto 语句跳过了x 的初始化。

小贴士：此示例也违背了 Rule 15.1。

```
{
    goto L1;
    uint16_t x = 10u;
L1:
    x = x + 1u;          /* 违规 - x 尚未赋值     */
}
```

参阅

Rule 15.1, Rule 15.3

Rule 9.2　集合或联合体的初始化应括在花括号“{}”中

C90 [Undefined 42], C99 [Undefined 76, 77]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则适用于对象和子对象的初始化。

形式为{0}的初始化程序将所有值都设置为 0，可用于初始化无嵌套括号的子对象。补充：不仅限子对象，只要是不使用嵌套括号的方式都可以，即使定义本身是有嵌套的，详见下面的示例。

小贴士：此规则本身并不要求显式初始化对象或子对象。

原理

使用花括号指示子对象的初始化可提高代码的清晰度，并迫使程序员考虑复杂数据结构(如多维数组或

结构数组)中元素的初始化。

例外

1. 可以使用字符串文字初始化数组。

2. 可以使用具有兼容结构或联合类型的表达式来初始化自动结构或联合。

3. 指定初始化可用于初始化子对象的一部分。

示例

以下三个初始化都是标准 C 所允许，且等效的初始化形式。此规则不允许使用第一种形式，因为它未使用花括号来显式标识子数组的初始化值。

```
int16_t y[3][2] = { 1, 2, 0, 0, 5, 6 };             /* 违规     */
int16_t y[3][2] = { { 1, 2 }, { 0 }, { 5, 6 } };    /* 合规     */
int16_t y[3][2] = { { 1, 2 }, { 0, 0 }, { 5, 6 } }; /* 合规     */
```

下面的示例中，使用了指定初始化来初始化子对象 z1[1]，通过例外 3，z1 初始化合规。基于相同的理由，z2的初始化亦合规。但是，z3 的初始化违规，因为子对象 z3[1]的一部分使用了指定初始化方式，但并未用花括号括起来。对 z4 的初始化是合规的，它使用了指定初始化来初始化子对象 z4[0]，并且将子对象 z4[1]的初始化用花括号括了起来。

```
int16_t z1[2][2] = { { 0 }, [1][1] = 1 };               /* 合规     */
int16_t z2[2][2] = { { 0 }, [1][1] = 1, [1][0] = 0};    /* 合规     */
int16_t z3[2][2] = { { 0 }, [1][0] = 0, 1 };            /* 违规     */
int16_t z4[2][2] = { [0][1] = 0, { 0, 1 } };            /* 合规     */
```

下例中的第一行在不使用嵌套括号的情况下初始化了 3 个子数组。 第二和第三行显示了编写相同初始化程序的等效方法。

```
/* 示例1，数组的初始化 */
float32_t a[3][2] = { 0 };                      /* 合规     */
float32_t a[3][2] = { { 0 }, { 0 }, { 0 } };    /* 合规     */
float32_t a[3][2] = { { 0.0f, 0.0f },
                      { 0.0f, 0.0f },
                      { 0.0f, 0.0f }
                    };                          /* 合规     */

/* 示例2，结构体的初始化 */
union u1 {
    int16_t   i;
    float32_t f;
} u = { 0 };                                    /* 合规     */
struct s1 {
    uint16_t len;
    char     buf[8];
```

```
} s[3] = {
    { 5u, { 'a', 'b', 'c', 'd', 'e', '\0', '\0', '\0' } },
    { 2u, { 0 } },
    { .len = 0u }    /* 合规 - 成员buf被隐式初始化    */
};                   /* 合规 - s[]完全初始化          */
```

Rule 9.3　数组不得部分初始化

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

如果数组对象或子对象的任何元素被显式初始化，则整个对象或子对象都应被显式初始化。

原理

为数组的每个元素提供一个显式的初始化，可以清晰地考虑到每个元素。

例外

1. {0}形式的初始化可用于显式初始化数组对象或子对象的所有元素。

2. 可以使用仅由指定的初始化组成的数组初始化形式，例如执行稀疏初始化。

3. 使用字符串文字初始化的数组不需要为每个元素都初始化。

示例

```
/* 合规 */
int32_t x[3] = { 0, 1, 2 };

/* 违规 - y[2] 被隐式初始化 */
int32_t y[3] = { 0, 1 };

/* 违规 - t[0] 和 t[3] 被隐式初始化 */
float32_t t[4] = { [1] = 1.0f, 2.0f };

/* 合规 - 以指定初始化方式进行矩阵的稀疏初始化 */
float32_t z[50] = { [1] = 1.0f, [25] = 2.0f };
```

下面的合规示例中，数组 arr 的每个元素均被显式初始化：

```
float32_t arr[3][2] =
{
    { 0.0f, 0.0f },
    { PI / 4.0f, -PI / 4.0f },
    { 0 }   /* 子对象arr[2]的所有元素均被初始化 */
};
```

下面示例中，数组元素 6 到 9 被隐式初始化为‘\0’：

```
char h[10] = "Hello"; /* 合规，符合例外 3 */
```

Rule 9.4　数组的元素不得被初始化超过一次

级别　必要

分析　可判定，单一编译单元

适用　C99

展开

此规则适用于对象和子对象的初始化。

通过在 C99 中提供的指定初始化，可以在初始化程序列表中初始化集合(结构或数组)或联合的组件的命名，并允许通过指定数组索引或结构以任何顺序初始化对象的元素，它们应用于的成员名称(不具有初始化值的元素采用未初始化对象的默认值)。

原理

使用指定初始化的方式进行初始化要格外小心，对象元素的初始化可能会不经意的重复，从而导致先前初始化的元素被覆盖。C99 标准并未指定是否覆盖重写的初始化副作用，且附件 J 中未列出。

为了允许稀疏的数组和结构，仅初始化应用程序所需的那些(元素或成员)是可接受的。

示例

数组初始化：

```
/*
 * 使用位置初始化的必须行为
 * 合规 - a1 被初始化为 -5, -4, -3, -2, -1
 */
int16_t a1[5] = { -5, -4, -3, -2, -1 };

/*
 * 使用指定初始化的类似行为
 * 合规 - a2 被初始化为 -5, -4, -3, -2, -1
 */
int16_t a2[5] = { [0] = -5, [1] = -4, [2] = -3, [3] = -2, [4] = -1 };

/*
 * 重复的指定初始值将覆盖先前的初始值设
 * 违规 - a3 被初始化为 -5, -4, -2, 0, -1
 */
int16 t a3[5] = { [0] = -5, [1] = -4, [2] = -3, [2] = -2, [4] = -1 };
```

在下面的违规示例中，不确定是否会产生副作用：

```
uint16_t *p;
void f(void)
{
    uint16_t a[2] = { [0] = *p++, [0] = 1 };
}
```

结构体初始化：

```
struct mystruct
{
    int32_t a;
    int32_t b;
    int32_t c;
    int32_t d;
};

/*
 * 使用位置初始化的必须行为
 * 合规 - s1 为 100, -1, 42, 999
 */
struct mystruct s1 = { 100, -1, 42, 999 };

/*
 * 使用指定初始化的类似行为
 * 合规 - s2 为 100, -1, 42, 999
 */
struct mystruct s2 = { .a = 100, .b = -1, .c = 42, .d = 999 };

/*
 * 重复的指定初始值将覆盖先前的初始值设
 * 违规 - s3 为 42, -1, 0, 999
 */
struct mystruct s3 = { .a = 100, .b = -1, .a = 42, .d = 999 };
```

Rule 9.5 在使用指定初始化方式初始化数组对象的情况下，应明确指定数组的大小

级别 必要

分析 可判定，单一编译单元

适用 C99

展开

该规则同样适用于作为灵活数组成员的数组子对象。

原理

如果一个数组未明确指定大小，则其大小由初始化元素的最高索引确定。当使用指定初始化时，可能

并不总是清楚哪个初始化具有最高的索引，尤其是在初始化包含大量元素的情况下。

为了清晰明确设计意图，应明确声明数组大小。这么操作也给程序开发过程中更改初始化数组元素索引提供了保护，因为它违反了限制(参见 C99 第 6.7.8 节)，在数组范围外初始化元素。

示例

```
/* 违规 - 可能无意中定义了一个只有一个元素的数组 */
int a1[] = { [0] = 1 };

/* 合规 */
int a2[10] = { [0] = 1 };
```

8.10 基本类型模型(The essential type model)

8.10.1 基本原理(Rationale)

本节中的规则共同定义了基本类型模型并限制了 C 语言的类型系统，以便：

1. 支持更强大的类型检查系统；
2. 定义规则为控制隐式和显式类型转换的使用提供合理依据；
3. 推广便携式编码做法；
4. 解决 ISO C 中发现的一些类型转换异常。

基本类型模型通过为被ISO C 认为属于算术类型的对象和表达式赋值基本类型来实现上述目的。例如，将 int 与char 相加得到的结果实质上具有字符类型，而不是整数提升实际产生的 int 类型。

附录 C 列出了基本类型模型的全部基本原理，附录D 则提供了所有算术表达式基本类型的全面定义。

8.10.2 基本类型(Essential type)

对象或表达式的基本类型由其基本类型类别和大小定义。

表达式的基本类型类别反映了其基本行为，它们可能是：

- 基本型为布尔值；
- 基本型为字符；
- 基本型为枚举值；
- 基本型为有符号；

- 基本型为无符号；
- 基本型为浮点值。

小贴士：每个枚举类型都是标识为 enum <i>的唯一基本枚举类型。这允许将不同的枚举类型处理为不同的类型，从而支持更强大的类型检查系统。一个例外是使用枚举类型在 C90 中定义布尔值，此类类型被认为具有基本布尔类型。另一个例外是使用附录 D 中定义的匿名枚举。匿名枚举是定义一组相关的常量整数的方式，被认为具有带符号的基本类型。

比较两个相同类型类别的类型时，术语“较宽”和“较窄”用于描述其相对大小(以字节为单位)。有时候两种不同的类型会以相同的大小实现。

下表显示了标准整数类型如何映射到基本类型类别。

基本类型分类					
布尔型	字符型	有符号型	无符号型	enum <i>	浮点型
_Bool	char	signed char signed short signed int signed long signed long long	unsigned char unsigned short unsigned int unsigned long unsigned long long	named enum	float double long double

小贴士：C99 实现可以提供扩展的整数类型，每种扩展类型将被分配一个与其等级和符号相符的位置(请参阅C99 第 6.3.1.1 节)。由本节中的规则强制执行的限制也适用于复合赋值运算符(例如^=)，因为它们等效于赋值通过使用算术，按位或移位运算符之一获得的结果。 例如：

```
u8a  += u8b  +1U;
```

等效于：

```
u8a = u8a + (u8b + 1U);
```

Rule 10.1 操作数不得为不适当的基本类型

C90 [Unspecified 23; Implementation 14, 17, 19, 32]

C99 [Undefined 13, 49; Implementation J3.4(2, 5), J3.5(5), J3.9(6)]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

在下表中，单元格中的数字表示在哪里限制将基本类型用作运算符的操作数。这些数字与下面“原理”

的 1-8 段相对应，并说明为什么要施加限制。

操作符	操作数	算术操作数的基本类型分类					
		布尔型	字符型	枚举型	有符号	无符号	浮点型
[ ]	整型数	3	4				1
+(正)		3	4	5			
-(负)		3	4	5		8	
+ -(加 减)	两侧	3		5			
* /(乘 除)	两侧	3	4	5			
%(取余数)	两侧	3	4	5			1
< > <= >=	两侧	3					
== !=	两侧						
! && ||	任意		2	2	2	2	2
<< >>	左侧	3	4	5,6	6		1
<< >>	右侧	3	4	7	7		1
~ & | ^	任意	3	4	5,6	6		1

操作符	操作数	算术操作数的基本类型分类					
		布尔型	字符型	枚举型	有符号	无符号	浮点型
?:	1st		2	2	2	2	2
?:	2nd,3rd						

在此规则下，“++”与“--”运算符的行为与双目运算符“+”、“-”相同。

其他规则进一步限制了可在表达式中使用的基本类型的组合。

原理

1 将浮点型表达式用于这些操作数，违反 C 语言标准。

2 当操作数被解释为布尔值时，应始终使用基本型为布尔类型的表达式。

3 如果操作数被解释为数值，则不应使用布尔类型的表达式。

4 如果操作数被解释为数值，则不应使用字符型的操作数。字符数据的数值是实现定义的，即编译环境不同，其值可能不同。

5 枚举型的操作数不应在算术运算中使用，因为枚举对象使用实现定义的整数类型，枚举型是否有符号依赖于编译环境。因此，涉及枚举对象的操作可能会产生意外类型的结果。请注意，来自匿名枚举的枚举常量的基本类型为有符号型。

6 移位运算符的左操作数和按位运算只能在无符号型的操作数上执行。它们在有符号型上使用所产生的数值是实现定义的，结果可能与预期不同。

7 移位运算符的右操作数应为无符号型，以确保不会因负移位而导致未定义的行为。

8 无符号型的操作数不应用作单目运算符“-”的操作数，因为结果的是否有符号由 int 的实现大小决定，亦即依赖编译环境。

例外

有符号型的常量表达式，若其值为非负，可以用作移位运算符的右手操作数。

示例

```
enum enuma { a1, a2, a3 } ena, enb; /* 基本型为枚举<enuma>型      */
enum { K1 = 1, K2 = 2 };            /* 基本型为有符号整形         */
```

下面的示例违规，注释中标识其违规的“原理”序号及其具体原因。

```
f32a & 2U       /* 原理 1 - 违反标准约束                    */
f32a << 2       /* 原理 1 - 违反标准约束                    */

cha && bla      /* 原理 2 - 字符型用作布尔型                */
ena ? a1 : a2   /* 原理 2 - 枚举型用作布尔型                */
s8a && bla      /* 原理 2 - 有符号型用作布尔型              */
u8a ? a1 : a2   /* 原理 2 - 无符号型用作布尔型              */
f32a && bla     /* 原理 2 - 浮点型用作布尔型                */

bla * blb       /* 原理 3 - 布尔型用作数值                  */
bla > blb       /* 原理 3 - 布尔型用作数值                  */

cha & chb       /* 原理 4 - 字符型用作数值                  */
cha << 1        /* 原理 4 - 字符型用作数值                  */

ena--           /* 原理 5 - 枚举型用于数值计算              */
ena * a1        /* 原理 5 - 枚举型用于数值计算              */

s8a & 2         /* 原理 6 - 有符号型用于按位运算的左操作数  */
50 << 3U        /* 原理 6 - 有符号型用于移位运算的左操作数  */

u8a << s8a      /* 原理 7 - 有符号型用于按位运算的右操作数  */
u8a << -1       /* 原理 7 - 有符号型用于按位运算的右操作数  */
```

```
-u8a            /* 原理 8 - 单目运算"-"不得用在无符号数上    */
```

下面示例既违反的此规则，又违反了 Rule 10.3：

```
ena += a1       /* 原理 5 - 枚举型用于数值计算               */
```

以下示例合规：

```
bla && blb
bla ? u8a : u8b

cha - chb
cha > chb

ena > a1
K1 * s8a        /* 合规 -  K1为匿名枚举，识别为有符号数      */

s8a + s16b
-(s8a) * s8b
s8a > 0
--s16b

u8a + u16b
u8a & 2U

u8a > 0U
u8a << 2U
u8a << 1        /* 合规 - 符合例外                           */

f32a + f32b
f32a > 0.0
```

下面示例符合此规则，但违反 Rule 10.2。

```
cha + chb
```

参 阅

Rule 10.2

Rule 10.2 字符类型的表达式不得在加减运算中使用不当

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

“正当使用”为：

1. 对于“+”运算符，一个操作数应为字符型，而另一个操作数应为有符号型或无符号型。 操作结果为字符型。

2. 对于“-”运算符，第一个操作数应为字符型，第二个操作数应为有符号型、无符号型或字符型。 如果两个操作数都是字符型，则其结果为标准类型(在这种情况下通常为 int)，否则结果为字符型。

原理

字符型(字符数据)的表达式不能做算术计算，因为该数据不代表数值。

允许上述用法，是因为它们可以潜在地合理处理字符数据。例如：

- 两个字符型的操作数相减可用于在范围从“ 0”到“ 9”的数字和相应的序数值之间进行转换；
- 可以用字符型和无符号型的加法来将序数值转换为“ 0”至“ 9”范围内的相应数字；
- 将字符型减去无符号型可用于将字符从小写转换为大写。

示例

下列示例合规：

```
'0' + u8a        /* 将 u8a 换算为数字(字符)                    */
s8a + '0'        /* 将 s8a 换算为数字(字符)                    */
cha - '0'        /* 将 cha 换算为序数(数值)                    */
'0' - s8a        /* 将 -s8a 换算为数字(字符)                   */
```

下列示例违规：

```
s16a - 'a'
'0' + f32a
cha + ':'
cha - ena
```

参 阅

Rule 10.1

Rule 10.3　表达式的值不得赋值给具有较窄基本类型或不同基本类型的对象

C90 [Undefined 15; Implementation 16]

C99 [Undefined 15, 16; Implementation 3.5(4)]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则涵盖以下操作：

1. 术语表(Glossary)中定义的任务；

2. 将 switch 语句的 case 标签中的常量表达式转换为控制表达式的类型提升。

原理

C 语言允许程序员具有极大的自由度，并允许自动执行不同算术类型之间的隐式赋值。但是，使用这些隐式转换可能会导致意外结果，并可能导致值、符号或精度的损失。有关 C 语言类型系统的更多详细信息， 请参见附录 C。

MISRA 的基本类型模型强制使用了更强的类型限制，从而降低了发生这些问题的可能性。

例外

1. 有符号型的对象的非负整数常量表达式可以赋值给无符号型的对象，只要它的值可以用该类型完全表示。

2. 初始化操作“{0}”可用于初始化聚合(数组、结构体)或联合类型。

示例

```
enum enuma { A1, A2, A3 } ena;
enum enumb { B1, B2, B3 } enb;
enum { K1=1, K2=128 };
```

合规示例：

```
uint8_t u8a = 0;                /* 符 合 例 外                    */
bool_t flag = (bool_t) 0;
bool_t set  = true;             /* true 为布尔型                  */
bool_t get  = (u8b > u8c);

ena = A1;
s8a = K1;       /* 常数值适合                                     */
u8a = 2;        /* 符合例外                                       */
u8a = 2 * 24;   /* 符合例外                                       */
cha += 1;       /* cha = cha + 1 字符赋值给字符                   */

pu8a = pu8b;                /* 相同的基本类型                     */
u8a  = u8b + u8c + u8d;     /* 相同的基本类型                     */
u8a  = (uint8_t) s8a;       /* 强制转换为相同的基本类型           */
u32a = u16a;                /* 赋值给更宽的基本类型               */
u32a = 2U + 125U;           /* 赋值给更宽的基本类型               */
use_uint16(u8a);            /* 赋值给更宽的基本类型               */
use_uint16(u8a + u16b);     /* 赋值给相同的基本类型               */
```

下面的示例违规，它们的基本类型不相同：

```
uint8_t u8a = 1.0f; /* 无符号(整)型与浮点型                       */
bool_t bla = 0;     /* 布尔型与有符号型                           */
```

```
cha  = 7;            /* 字符型与有符号型                          */
u8a  = 'a';          /* 无符号型与字符型                          */
u8b  = 1 - 2;        /* 有符号型与无符号型                        */
u8c += 'a';          /* u8c = u8c + 'a' 字符型与无符号型相加      */
use_uint32(s32a);    /* 有符号型与无符号型                        */
```

下面的示例违规，因为它们被赋值给较窄的基本类型：

```
s8a  = K2;                  /* 常数值超出被赋值的类型的值域范围    */
u16a = u32a;                /* uint32_t -> uint16_t                */
use_uint16(u32a);           /* uint32_t -> uint16_t                */
uint8_t foo1(uint16_t x)
{
    return x;               /* uint16_t -> uint8_t                 */
}
```

参阅

Rule 10.4, Rule 10.5, Rule 10.6

Rule 10.4　执行通常算术转换的运算符的两个操作数都具有相同的基本类型类别

C90 [Implementation 21], C99 [Implementation 3.6(4)]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

该规则适用于常规算术转换中描述的运算符(请参阅 C90 第6.2.1.5 节，C99 第6.3.1.8 节)。此规则适用所有双目运算符，除了移位(<< >>)、逻辑与(&&)、逻辑或(||)和逗号“,”运算符。

小贴士：此规则覆盖三元运算符的第二和第三操作数。

原理

C 语言允许程序员具有极大的自由度，并允许自动执行不同算术类型之间的隐式赋值。但是，使用这些隐式转换可能会导致意外结果，并可能导致值、符号或精度的损失。有关 C 语言类型系统的更多详细信息， 请参见附录 C。(这一段与 Rule 10.3 相同)

MISRA 的基本类型模型强制使用了更强的类型限制，以保证隐式类型转换的结果与开发人员的期望一致。

例外

以下操作被允许，用以保证常见的字符操作被识别为合规：

1. 双目运算符“+”和“+=”允许一个操作数是字符型，而另一个操作数为有符号型或无符号型；

2. 双目运算符“-”和“--”允许字符型的左操作数和有符号型或无符号型的右操作数。

示例

```
enum enuma { A1, A2, A3 } ena;
enum enumb { B1, B2, B3 } enb;
```

下面的合规示例中，运算符两侧的数据具有相同的基本类型：

```
ena > A1
u8a + u16b
```

下面示例符合例外 1：

```
cha += u8a
```

下面示例既不合此规则，也违反了 Rule 10.3：

```
s8a += u8a  /* 有符号与无符号数计算                */
```

下面示例违规：

```
u8b + 2     /* 无符号数与有符号数计算              */
enb > A1 /* 枚举<enumb>型 与 枚举<enuma>型     */
ena == enb /* 枚举<enuma>型 与 枚举<enumb>型 */
u8a += cha /* 无符号整型数与字符型数据 */
```

参阅

Rule 10.3, Rule 10.7

Rule 10.5 表达式的值不应(强制)转换为不适当的基本类型

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

展开

下表中显示了应避免的强制转换，表中第一列为强制转换(显示转换)的目标类型。

基本类型	源类型					
目标类型	布尔型	字符型	枚举型	有符号型	无符号型	浮点型
布尔型		避免	避免	避免	避免	避免
字符型	避免					避免
枚举型	避免	避免	避免*	避免	避免	避免
有符号型	避免					
无符号型	避免					

浮点型	避免	避免				

*：枚举型可以强制转换为枚举型，前提是强制转换为相同的基本枚举类型。这样的转换是多余的。即，应避免将一种枚举型转换为其他枚举型的操作。

不得将 void 型强转为其他任何基本类型，因为该操作会导致不确定的行为。Rule 1.3 中涵盖此限制。

原理

出于合法功能的原因，可能会引入显式强制转换，例如：

- 更改执行后续算术运算的类型；
- 故意截断值；
- 为了清楚起见，使类型转换更清晰。

但是，某些显式强制转换被一般是不合适的：

- 在 C99 中，对_Bool 进行强制转换或赋值的结果始终为 0 或 1。强制转换为基本型为布尔型的其他类型的操作，其实并不必要。
- 强制转换为枚举型可能会导致其值不属于该类型的枚举常量集；
- 从布尔型转换为其他任何类型都不太有意义；
- 浮点型和字符型之间的转换没有意义，因为两种表示形式之间没有精确的映射。

例外

可以将带符号的值为 0 或 1 的整数常量表达式强制转换为布尔型。 这允许实现非C99 布尔模型。

示例

```
(bool_t) false  /* 合规 - C99标准中'false'是布尔型 */
(int32_t) 3U    /* 合规                            */
(bool_t) 0      /* 合规 - 符合例外                 */
(bool_t) 3U     /* 违规                            */
(int32_t) ena   /* 合规                            */
(enum enuma) 3  /* 违规                            */
(char) enc      /* 合规                            */
```

参阅

Rule 10.3, Rule 10.8

8.10.3 复合运算符和表达式

附录 C 中提到的某些问题，可以通过限制可能应用于非平凡表达式的隐式和显式转换来避免。 这些问题包括：

- 计算整数表达式的类型的困惑：表达式的基本类型取决于任何整数提升后的操作数的类型。算术运算结果的类型取决于 int 的实现大小；
- 程序员之间普遍存在的误解：进行计算的类型受赋值或强制转换结果的类型的影响。 错误的期望可能会导致意想不到的结果。

除了先前的规则外，基本类型模型还对操作数为复合表达式的表达式进行了进一步的限制，如下所述。以下内容在本文档中定义为复合运算符：

- 乘除法(*，/，%)
- 加减法(双目运算符“+”，双目运算符“-”)
- 按位操作(&，|，^)
- 移位操作(<<，>>)
- 条件运算(?:)，如果第二个或第三个操作数是复合表达式

复合赋值等效于将复合表达式的结果进行赋值(赋值)。

复合表达式在本文档中定义为非常量表达式(复合运算的结果)。

小贴士：

- *复合赋值运算符的结果不是复合表达式；*
- *带括号的复合表达式也是复合表达式；*
- *常量表达式不是复合表达式。*

Rule 10.6　复合表达式的值不得赋值给具有较宽基本类型的对象

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则涵盖 Rule 10.3 中描述的赋值操作。

原理

基本原理在关于复合运算符和表达式的介绍中进行了描述(请参见第 8.10.3 节)。

示例

合规示例：

```
u16c = u16a + u16b;              /* 相同的基本类型                    */
u32a = (uint32_t)u16a + u16b;    /* 强制转换，表达式为uint32_t型加法 */
```

违规示例：

```
u32a = u16a + u16b;              /* 赋值时隐式转换                    */
use_uint32(u16a + u16b);         /* 函数参数的隐式转换                */
```

参阅

Rule 10.3, Rule 10.7, 章节 8.10.3

Rule 10.7 如果将复合表达式用作执行常规算术转换的运算符的一个操作数，则另一个操作数不得具有更宽的基本类型

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

基本原理在关于复合运算符和表达式的介绍中进行了描述(请参见第 8.10.3 节)。

限制复合表达式的隐式转换意味着，表达式中的算术运算序列必须以完全相同的基本类型进行。 这减少了可能发生的开发者困惑。

小贴士：这并不意味着表达式中的所有操作数都必须具有相同的基本类型。

表达式 u32a +u16b +u16c 是合规的，因为两个加法都将在类型uint32_t 中执行。 在这种情况下，被隐式转换的是非复合表达式。即：u16b 和u16c 在计算中被转换为uint32_t。

表达式(u16a + u16b)+ u32c 是违规的，因为名义上在 uint16_t 类型中执行了左加法，而在uint32_t 类型中进行了右加法，因此需要将复合表达式 u16a + u16b 隐式转换为 uint32_t。

示例

合规示例：

```
 u32a  * u16a  + u16b              /* 无表达式转换          */
(u32a  * u16a) + u16b             /* 无表达式转换          */
 u32a  * ((uint32_t) u16a + u16b) /* * 两侧操作数类型一致 */
 u32a += (u32b + u16b)            /* 无表达式转换          */
```

违规示例：

```
u32a  * (u16a + u16b)   /* 表达式(u16a + u16b)被隐式转换    */
u32a += (u16a + u16b)   /* 表达式(u16a + u16b)被隐式转换    */
```

参阅

Rule 10.4, Rule 10.6, 章节 8.10.3

Rule 10.8　复合表达式的值不得转换为其他基本类型或更宽的基本类型

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

基本原理在关于复合运算符和表达式的介绍中进行了描述(请参见第 8.10.3 节)。

不允许强制转换为更宽的基本类型，是因为不同的实现结果可能不同。考虑下面表达式：

```
(uint32_t)(u16a + u16b);
```

在 16 位机上，加法将以 16 位执行，舍弃高于 16 位的值作为结果，然后转换为 32 位。 但是，在 32 位机上，加法将以 32 位进行，并且将保留在 16 位机上丢失的高阶位。

强制转换为具有相同基本类型的较窄类型是可以接受的，因为结果的显式截断导致相同的信息丢失。

示例

```
(uint16_t)(u32a + u32b)      /* 合规                              */
(uint16_t)(s32a + s32b)      /* 违规 - 不同的基本类型             */
(uint16_t)s32a               /* 合规 - s32a并非复合表达式         */
(uint32_t)(u16a + u16b)      /* 违规 - 强转为更宽的类型           */
```

参阅

Rule 10.5, 章节 8.10.3

8.11 指针类型转换(Pointer type conversions)

指针类型可以分类如下：

- 指向对象的指针；
- 函数指针；
- 指向不完整类型对象的指针；
- 指向 void 的指针；
- 空指针常量，即值 0，可以选择强制转换为 void *。

C 语言标准允许的涉及指针的转换仅限于：

- 从指针类型到 void 的转换；
- 从指针类型到算术类型的转换；
- 从算术类型到指针类型的转换；
- 从一种指针类型到另一种指针类型的转换。

尽管 C 语言标准中并未禁止，但指针与整数型意外的任何算数类型间的转换都是未定义的。以下被允许的指针转换不需要强制显式转换：

- 从指针类型到_Bool 的转换(仅 C99)；
- 从空指针常量到指针类型的转换；
- 在目标类型具有源类型的所有类型限定符的前提下，从一种指针类型转换为与其兼容的指针类型；
- 在目标类型具有源类型的所有类型限定符的前提下，在指向(完整类型)对象或不完整类型的指针与“void *”型或其限定版本的指针之间进行转换。

在 C99 中，任何不属于该指针转换子集的隐式转换都将违反约束(C99 第 6.5.4 节和第 6.5.1 节)。在 C90 中，任何不属于指针转换子集的隐式转换都会导致未定义的行为(C90 第6.3.4 节和第 6.3.1 节)。指针类型和整数类型之间的转换是实现定义的(依赖于编译环境设置)。

Rule 11.1　不得在指向函数的指针和任何其他类型的指针之间进行转换

C90 [Undefined 24, 27 - 29], C99 [Undefined 21, 23, 39, 41]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

指向函数的指针仅可与指向兼容类型的函数的指针间相互转换。

原理

指向函数的指针与以下任何一种的转换，都将导致不确定的行为：

- 指向对象的指针；
- 指向不完整类型的指针；
- void *。

若通过与被调用函数类型不兼容的指针来调用函数，其行为是不确定的。C 标准允许将指向函数的指针转换为具有不同类型的函数的指针和将整数转换为指向函数的指针。但是，这两项都被此规则禁止，以避免使用不兼容的指针类型调用函数而导致的未定义行为。

例外

1. 空指针常量(NULL)可以转换为指向函数的指针；

2. 指向函数的指针可以转换为 void；

3. 函数类型(标识符)可以隐式转换为指向该函数类型的指针。

小贴士：例外 3 涵盖了C90 第6.2.2.1 节和C99 第6.3.2.1 节中描述的隐式转换。 这些转换通常发生在以下情况：

- *直接调用函数，即：使用函数标识符表示要调用的函数；*
- *将函数赋值给函数指针。*

示例

```
typedef void (*fp16) (int16_t n);
typedef void (*fp32) (int32_t n);

#include <stdlib.h>                 /* 获取宏 NULL                          */

fp16 fp1 =  NULL;                   /* 合规 - 例外 1                        */
fp32 fp2 = (fp32)fp1;               /* 违规 - 前后函数指针指向类型不同      */

if (fp2 != NULL)                    /* 合规 - 例外 1                        */
{
}

fp16 fp3 = (fp16)0x8000;            /* 违规 - 整型数转换为指针              */

fp16 fp4 = (fp16)1.0e6F;            /* 违规 - 浮点数转换为指针              */
```

在以下示例中，函数调用返回指向函数类型的指针。 将返回值强制转换为void 符合此规则。

```
typedef fp16 (*pfp16)(void);

pfp16 pfp1;

(void)(*pfp1());                    /* 合规 - 例外 2 - 函数指针换行为void   */
```

以下示例显示了从函数类型(标识符)到指向该函数类型的指针的合规的隐式转换：

```
extern void f(int16_t n);

f(1);                           /* 合规 - 例外 3, 函数f向函数指针的隐式转换 */

fp16 fp5 = f;                   /* 合规 - 例外 3                            */
```

Rule 11.2　不得在指向不完整类型的指针和其他任何类型间进行转换

C90 [Undefined 29], C99 [Undefined 21, 22, 41]

级别　必要

分析　可判定，单一编译单元

适用　　C90，C99

展开

指向不完整类型的指针不得转换为其他类型。

不得将其他类型转换为指向不完整类型的指针。

此规则不适用于指向 void 的指针，尽管指向 void 的指针也是指向不完整类型的指针，它们将被 Rule 11.5 涵盖。

原理

普通指针与指向不完整类型间的转换可能会导致指针未正确对齐，从而导致行为不确定。

将不完整类型的指针转与浮点型互相转换，也会导致未定义的行为。

指向不完整类型的指针有时用作隐藏对象的表示形式的一种封装方式。将不完整类型的指针转换为指向对象(完整类型/明确类型)的指针会破坏这种封装。

例外

1. 空指针常量(NULL)可以转换为指向不完整类型的指针。

2. 指向不完整类型的指针可能会转换为 void。

示例

```
struct s;                   /* 不完整类型1              */
struct t;                   /* 不完整类型2              */
struct s *sp;
struct t *tp;
int16_t  *ip;

#include <stdlib.h>         /* 获取宏 NULL              */

ip = (int16_t  *)sp;        /* 违规                     */
sp = (struct s *)1234;      /* 违规                     */
tp = (struct t *)sp;       /* 违规 - 不完整类型间转换  */
sp = NULL;                  /* 合规 - 例外 1            */
/*                                                      */

struct s *f(void);
(void)f();                  /* 合规 - 例外 2            */
```

参 阅

Rule 11.5

Rule 11.3　不得在指向不同对象类型的指针之间执行强制转换

C90 [Undefined 20], C99 [Undefined 22, 34]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则适用于指针所指向的非限定类型。

原理

指向不同对象的指针间的转换可能会导致指针未正确对齐，从而导致未定义的行为。

即使已知转换会产生正确对齐的指针，使用该指针访问对象的行为也可能是未定义的。例如，如果将类型为 int 的对象作为 short 进行访问，则即使 int 和 short 具有相同的表示和对齐要求，其行为也是未定义的。 有关详细信息，请参见 C90 第6.3 节，C99 第6.5 节第 7 段。

例外

将指向对象类型的指针转换为指向对象类型为 char，signed char 或 unsigned char 之一的指针是被允许的。C 标准确保了可以使用指向这些类型的指针来访问对象的各个字节。

示例

```
uint8_t  *p1;
uint32_t *p2;

/* 违规 - possible incompatible alignment */
p2 = (uint32_t *)p1;

extern uint32_t read_value(void);
extern void print(uint32_t n);

void f (void)
{
    uint32_t u     = read_value();
    uint16_t *hi_p = (uint16_t *)&u;    /* 违规 - 即使可能正确对齐了 */

    *hi_p = 0;       /* 尝试在大端模式的机器上清除高16位              */
    print(u);        /* 上一行代码可能不被执行                        */
}
```

以下示例符合要求，因为此规则适用于不合格的(unqualified,翻译为“无限制条件的”可能更合适，

下同)指针类型。它不会阻止将类型限定符添加到对象类型。

```
const short *p;
const volatile short *q;

q = (const volatile short *)p;  /* 合规 */
```

以下示例是不合规的，因为不合格的指针类型不同，即“指向 const 限定的int 的指针”和“指向 int 的指针”。

```
int * const * pcpi;
const int * const * pcpci;

pcpci = (const int * const *)pcpi;
```

参阅

Rule 11.4, Rule 11.5, Rule 11.8

Rule 11.4　不得在指向对象的指针和整数类型之间进行转换

C90 [Undefined 20; Implementation 24]

C99 [Undefined 21, 34; Implementation J.3.7(1)]

级别　建议

分析　可判定，单一编译单元

适用　C90，C99

展开

指针不应转换为整型数。

整型数也不应转换为指针。

原理

将整数转换为指向对象的指针可能会导致指针未正确对齐，从而导致未定义的行为。

将对象的指针转换为整数可能会产生无法以所选整数类型表示的值，从而导致未定义的行为。

小贴士：*C99 类型 intptr_t 和 uintptr_t(在<stdint.h>中声明)分别表示有符号和无符号整数类型，且能够表示指针的值。尽管如此，此规则仍不允许在对象的指针与这些类型之间进行转换，因为它们的使用不能避免与未对齐的指针相关联的不确定行为。*

在可能的情况下，应避免在指针和整数类型之间进行强制转换，但在寻址存储器映射的寄存器或其他特定于硬件的功能时(这种强制转换)可能有必要。如果使用整数和指针之间的转换，则应注意确保产生的任何指针都不会引起规则 11.3 中讨论的未定义行为。

例外

具有整数类型的空指针常量(NULL)可以转换为指向对象的指针。

示例

```
uint8_t *PORTA = (uint8_t *)0x0002;      /* 违规 */

uint16_t *p;

int32_t   addr = (in t32_t)&p;           /* 违规 */
uint8_t  *q    = (uint8_t *)addr;        /* 违规 */
bool_t    b    = (bool_t)p;              /* 违规 */

enum etag { A, B } e = (enum etag)p;     /* 违规 */
```

参阅

Rule 11.3, Rule 11.7, Rule 11.9

Rule 11.5　不得将指向 void 的指针转换为指向对象的指针

C90 [Undefined 20], C99 [Undefined 22, 34]

级别　建议

分析　可判定，单一编译单元

适用　C90，C99

原理

将指向 void 的指针转换为指向对象的指针可能会导致指针未正确对齐，从而导致未定义的行为。这种转换应尽可能避免使用，但其也可能有必要使用，例如在使用内存分配功能处理时。如果使用从对象的指针到 void 的指针转换，则应确保确保生成的任何指针都不会引起规则 11.3 中讨论的未定义行为。

例外

类型为指向 void 的指针的空指针常量(NULL)可以转换为对象的指针。

示例

```
uint32_t *p32;
void     *p;
uint16_t *p16;

p   = p32;              /* 合规 - 指向uint32_t的指针转换为指向void的指针 */
p16 = p;                /* 违规                                          */
p   = (void *)p16;      /* 合规                                          */
p32 = (uint32_t *)p;    /* 违规                                          */
```

参阅

Rule 11.2, Rule 11.3

Rule 11.6　不得在指向 void 的指针和算术类型之间执行强制转换

C90 [Undefined 29; Implementation 24]

C99 [Undefined 21, 41; Implementation J.3.7(1)]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

将整数转换为指向void 的指针可能会导致指针未正确对齐，从而导致未定义的行为。

将指向 void 的指针转换为整数可能会产生无法以所选整数类型表示的值，从而导致未定义的行为。任何非整数算术类型和指向 void 的指针之间的转换都是未定义的。

例外

可以将值为 0 的整数常量表达式转换为指向 void 的指针。此即空指针常量 NULL 在代码中的定义形式。

示例

```
void     *p;
uint32_t  u;

/* 违规 - 行为由实现定义(受编译环境及其设置影响)     */
p = (void *)0x1234u;

/* 违规 - 未定义行为                                 */
p = (void *)1024.0f;

/* 违规 - 行为由实现定义(受编译环境及其设置影响)     */
u = (uint32_t) p;
```

Rule 11.7　不得在指向对象的指针和非整数算术类型之间执行强制转换

C90 [Undefined 29; Implementation 24]

C99 [Undefined 21, 41; Implementation J.3.7(1)]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则中的“非整数算术类型”表示以下类型之一：

- 基本类型是布尔型；
- 基本类型是字符型；
- 基本类型是枚举型；
- 基本类型是浮点型。

原理

将基本型为布尔型，字符型或枚举型的数据转换为指向对象的指针可能会导致指针未正确对齐，从而导致未定义的行为。

将对象的指针转换为布尔型，字符型或枚举型可能会产生无法用所选整数类型表示的值，从而导致未定义的行为。

任何指向对象的指针与浮点型之间的转换，都将导致未定义的行为。

示例

```
int16_t   *p;
float32_t  f;

f = (float32_t)p; /* 违规 */
p = (int16_t *)f; /* 违规 */
```

参 阅

Rule 11.4

Rule 11.8　强制转换不得从指针指向的类型中删除任何 const 或volatile 限定符

C90 [Undefined 12, 39, 40], C99 [Undefined 30, 61, 62]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

通过强制转换来删除与所指向类型相关的限定条件的任何尝试都违反了类型限定原则。

小贴士：此处提到的限定条件与可应用于指针本身的任何限定条件均不同。

如果从所寻址的对象中删除限定符，可能会出现下述问题：

- 删除 const 限定符可能会绕过对象的只读属性并导致其被修改；
- 访问对象时，删除 const 限定符可能会导致异常。
- 删除 volatile 限定符可能会导致无法访问已优化的对象。

小贴士：删除 C99 中的 restrict(限制类型)限定符是无害的。

示例

```
        uint16_t x;
        uint16_t * const cpi = &x; /* 指针常量                        */
        uint16_t * const *pcpi;    /* 指向指针常量的指针              */
        uint16_t *       *ppi;
   const uint16_t        *pci;     /* 指向常量的指针                  */
volatile uint16_t        *pvi;     /* 指向 volatile 的指针            */
        uint16_t         *pi;

pi  = cpi;                         /* 合规 - 无隐式转换，无强制转换   */
pi  = (uint16_t *)pci;             /* 违规                            */
pi  = (uint16_t *)pvi;             /* 违规                            */
ppi = (uint16_t * *)pcpi;          /* 违规                            */
```

参 阅

Rule 11.3

Rule 11.9　宏“NULL”是整数型空指针常量的唯一允许形式

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

NULL 出现在以下应用场景中时，应将宏 NULL 扩展得出一个值为 0 的整数常量表达式：

- 将值赋值给指针；
- 作为“==”或“!=”运算符的操作数，其另一个操作数是指针；
- 作为“?:”运算符的第二个操作数，其第三个操作数是一个指针；
- 作为“?:”运算符的第三个操作数，其第二个操作数是一个指针。

忽略空格和任何括号，任何这样的整数常量表达式都将代表 NULL 的整个扩展。

*小贴士：允许使用(void *)0 形式的空指针常量，无论其是否是从NULL 扩展而来。*

原理

使用 NULL 而不是 0 可以清楚地表示使用空指针常量(的场景)。

示例

在以下示例中，p2 的初始化是合规的，因为整数常量表达式 0 未出现在此规则禁止的场景中。

```
int32_t *p1 = 0;            /* 违规 */
int32_t *p2 = (void *)0;    /* 合规 */
```

在下面的示例中，p2 和(void *)0 之间的比较是合规的，因为整数常量表达式 0 作为强制转换的操作数出现，而不是在此规则禁止的场景中出现。

```
#define MY_NULL_1  0
#define MY_NULL_2 (void *)0

if (p1 == MY_NULL_1) /* 违规 */
{
}
if (p2 == MY_NULL_2) /* 合规 */
{
}
```

以下示例合规，因为使用实现提供的宏 NULL 是被允许的，即使该宏扩展为值为 0 的整数常量表达式也是如此。

```
/* 也可以是 stdio.h, stdlib.h 或其他头文件 */
#include <stddef.h>

extern void f(uint8_t *p);

/* 任何NULL的合理定义都是合规的，例如:
* 0
* (void *)0
* (((0)))
* (((1 - 1)))
*/
```

```
f(NULL);
```

参 阅

Rule 11.4

8.12 表达式(Expressions)

Rule 12.1 表达式中运算符的优先级应明确

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

展开

下表用于此规则的定义。

描述	运算符或操作数	优先级
主表达式(Primary)	标识符，常量，字符串，(表达式)	16(高)
后缀运算(Postfix)	[] ()(函数调用) . -> ++(后自增) --(后自减) (){} (C99:复合文字)	15
单目/前缀运算(Unary)	++(前自增) --(前自减) & * + - ~ ! sizeof defined(预编译)	14
强制转换(Cast)	()	13
乘除法(Multiplicative)	* / %	12
加减法(Additive)	+ -	11
移位(Bitwise shift)	<< >>	10
大小比较(Relational)	< > <= >=	9
相等比较(Equality)	== !=	8
按位与(Bitwise AND)	&	7
按位异或(Bitwise XOR)	^	6
按位或(Bitwise OR)	|	5
逻辑与(Logical AND)	&&	4
逻辑或(Logical OR)	||	3

条件运算(Conditional)	?:	2
赋值(Assignment)	= *= /= %= += -= <<= >>= &= ^= |=	1
逗号(Comma)	,	0(低)

我们会选择此表中使用的优先级对此规则进行简洁的描述。 它们不一定与其他(可能遇到的)运算符优先级描述中相同。

就此规则而言，表达式的优先级是该表达式在分析树的根处元素(操作数或运算符)的优先级。

例如：表达式a << b + c 的解析树可以表示为：

此分析树的根元素为“ <<”，因此表达式的优先级为 10。

此规则提供以下建议：

- sizeof 运算符的操作数应放在括号内；
- 优先级在 2 到12 范围内的表达式应在满足下述两种情况的操作数周围加上括号：
  - 优先顺序小于 13，并且
  - 优先级大于表达式的优先级。

原理

C 语言的运算符数量很多，并且它们的相对优先级并不直观。这会导致经验不足的程序员犯错误。使用括号使运算符优先级明确，可以消除程序员的期望与事实不符的可能性。这也使代码的审阅者或维护者可以清楚地了解程序员的原始意图。

众所周知，括号的过度使用会使代码混乱，并降低其可读性。此规则旨在在难以理解的代码之间达成折衷，因为其中包含太多或更少的括号。

示例

下面的示例显示具有单目或后缀运算符的表达式，其操作数是主表达式或顶级运算符具有优先级 15 的表达式。

```
a[i]->n;        /* 合规 - 无需写成 (a[i])->n                    */
*p++;           /* 合规 - 无需写成 *(p++)                       */
sizeof x + y;   /* 违规 - 应写成 sizeof(x) + y 或 sizeof(x + y) */
```

以下示例显示了包含具有相同优先级的运算符的表达式。 所有这些都是合规的，但取决于 a，b 和 c 的类型，具有多个运算符的任何表达式都可能违反其他规则。

```
a + b;
a + b + c;
(a + b) + c;
a + (b + c);
a + b - c + d;
(a + b) - (c + d);
```

以下示例列举了各种混合运算符表达式：

```
/* 合规 - 无需写成 f((a + b), c) */
x = f (a + b, c);

/* 违规
 * 条件运算符(优先级2)的操作数为:
 * == 优先级8  需要括号
 * a  优先级16 不需要括号
 * -  优先级11 需要括号
 */
x = a == b ? a : a - b;

/* 合规 */
x = (a == b) ? a : (a - b);

/* 合规
 * << 运算符(优先级10)的操作数为：
 * a   优先级16 不需要括号
 * (E) 优先级16 已经有括号
 */
x = a << (b + c);

/* 合规
 * && 运算符(优先级4)的操作数为:
 * a 优先级16 不需要括号
 * && 优先级4 优先级相同 不需要括号
 */
if (a && b && c)
{
}

/* 合规
 * && 运算符(优先级4)的操作数为:
 * defined(X) 优先级14 不需要括号
 * (E)        优先级16 已经有括号
 */
#if defined(X) && ((X + Y) > Z)
```

```
/* 合规
* && 运算符(优先级4)的操作数为:
* !defined(X) 优先级14 不需要括号
*  defined(Y) 优先级14 不需要括号
* ! 运算符(优先级14)的操作数为:
* defined(X) 优先级14 优先级相同 不需要括号
*/
#if !defined(X) && defined(Y)
```

小贴士：此规则不要求将逗号(,)运算符的操作数括起来。 Rule 12.3 禁止使用逗号(,)运算符。

```
x = a, b;   /* 合规 - 解析为 (x = a), b */
```

Rule 12.2 移位运算符的右操作数应在零到比左操作数基本类型的位宽度小一的范围内

C90 [Undefined 32], C99 [Undefined 48]

级别 必要

分析 不可判定，系统范围

适用 C90，C99

原理

右操作数为负或大于或等于左操作数的宽度的行为未定义。

举例来说，如果左移或右移的左操作数是 16 位整数，则确保仅将其移位 0 到 15 之间的数字非常重要。

有关移位运算符的基本类型和基本类型的限制的说明，请参见第 8.10 节。

我们有多种方法可以确保遵守此规则。最简单的方法是将右操作数设为常数(然后可以静态检查其值)。另一种方法是使用无符号整数类型以确保操作数为非负数，然后仅需要检查上限(在运行时动态检查或通过评审(review)检查)。否则，将需要检查两个限制。

示例

```
u8a  = u8a << 7;            /* 合规 */
u8a  = u8a << 8;            /* 违规 */
u16a = (uint16_t)u8a << 9;  /* 合规 */
```

为便于理解以下示例，应注意，1u 的基本类型是 unsigned char，而 1UL 的基本类型是 unsigned long。

```
1u << 10u;                  /* 违规 */
(uint16_t)1u << 10u;        /* 合规 */
1UL << 10u;                 /* 合规 */
```

Rule 12.3 不得使用逗号(,)运算符

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

原理

使用逗号运算符通常不利于代码的可读性，并且通常可以通过其他方式实现相同的效果。

示例

```
f((1, 2), 3);   /* 违规 - 究竟有几个形参? */
```

以下示例违背了此规则和其他规则：

```
for (i = 0, p = &a[0]; i < N; ++i, ++p)
{
}
```

Rule 12.4 常量表达式的求值不应导致无符号整数的回绕

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

展开

此规则适用于满足常量表达式约束的表达式，无论它们是否出现在需要常量表达式的上下文中。

如果未对表达式求值(例如，它出现在逻辑 AND 运算符的右操作数中，而其左操作数始终为 false)，则此规则不适用。

原理

无符号整数表达式不会严格溢出，而是会回绕。 尽管在运行时使用模运算可能有充分的理由，但是在编译时故意使用模运算的可能性较小。

示例

与 case 标签关联的表达式必须是常量表达式。如果在对 case 表达式求值时发生无符号回绕，则可能是无意的。在具有 16 位宽度的 int 类型的计算机上，在下面的示例中，任何大于或等于 65024 的 BASE 值都将导致回绕：

```
#define BASE 65024u

switch(x )
{
    case BASE + 0u:
        f();
        break;
```

```
    case BASE + 1u:
        g();
        break;
    case BASE + 512u:   /* 违规 - 会回绕回 0 */
        h();
        break;
}
```

预处理指令#if 和#elif 的控制表达式必须是常量表达式。

```
#if 1u + (0u - 10u)     /* 违规 -  因为 (0u - 10u) 会回绕 */
```

在下面这个示例中，表达式 DELAY + WIDTH 的值为 70000，但是在具有 16 位 int 类型的计算机上，该值将回绕为 4464。

```
#define DELAY 10000u
#define WIDTH 60000u

void fixed_pulse(void)
{
    uint16_t off_time16 = DELAY + WIDTH;    /* 违规 */
}
```

在下面的合规示例中，此规则不适用于表达式 c +1，因为它访问了对象，因此不满足常量表达式的约束：

```
const uint16_t c = 0xffffu;

void f(void)
{
    uint16_t y = c + 1u;    /* 合规 */
}
```

在下面的示例中，子表达式(0u-1u)导致无符号整数环绕。在 x 的初始化中，不评估子表达式，因此表达式是合规的。但是，在y 的初始化中，可能会对其进行求值，因此表达式违规。

```
bool_t b;

void g(void)
{
    uint16_t x = (0u == 0u) ? 0u : (0u - 1u);   /* 合规 */
    uint16_t y = b ? 0u : (0u - 1u);            /* 违规 */
}
```

8.13 副作用(Side effects)

Rule 13.1 初始化程序列表不得包含持久性副作用

C99 [Unspecified 17, 22]

级别 必要

分析 不可判定，系统范围

适用 C99

原理

C90 标准对具有聚合类型的自动对象的初始化程序有约束，使其仅包含常量表达式。但 C99 标准则允许自动聚合初始化程序包含在运行时评估的表达式。它还允许复合文字充当匿名初始化的对象。在初始化程序列表中的表达式求值过程中，副作用的发生顺序是不确定的，因此，如果这些副作用持续存在，则初始化行为是不可预测的。

示例

```
volatile u int16_t v1;

void f(void)
{
    /* 违规 - 易失性访问是持久的副作用  */
    uint16_t a[2] = { v1, 0 };
}

void g(uint16_t x, uint16_t y)
{
    /* 合规 - 无副作用                  */
    uint16_t a[2] = { x + y, x - y };
}

uint16_t x = 0u;

extern void p(uint16_t a[2]);

void h(void)
{
    /* 违规 - 两个副作用                */
    p((uint16_t[2]) { x++, x++ });
}
```

参阅

Rule 13.2 在所有合法的评估命令下，表达式的值应与其持续的副作用相同

C90 [Unspecified 7－9; Undefined 18], C99 [Unspecified 15－18; Undefined 32]

级别 必要

分析 不可判定，系统范围

适用 C90，C99

展开

在任何两个相邻序列点之间或在任何完整表达式内：

1. 不得修改任何对象超过一次；
2. 不得修改和读取对象，除非对对象值的任何此类读取对计算要存储到对象中的值有帮助；
3. 最多只能有一个易失性(volatile)限定类型的修改访问权限；
4. 易失性(volatile)限定类型的读取访问不得超过一个。

小贴士：可以通过指针或调用的函数间接访问对象，也可以通过表达式直接访问对象。

小贴士：此展开有意描述的比规则的标题严格。作为其结果，表达式：

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

即使 x 的值和其持久副作用(如果 x 是易失性(volatile)类型)与评估或副作用的顺序无关，此规则也不允许这样做。

序列点在 C90 和C99 标准的附录 C 中进行了总结。 C90 中的序列点是 C99 中的序列子集。

C90 标准的 6.6 节和 C99 标准的 6.8 节定义了完整的表达式。

原理

该标准在评估表达式时为编译器提供了相当大的灵活性。 大多数运算符可以按任何顺序对其操作数求值。 主要的例外是：

- 逻辑与(&&)，仅当第一个操作数的取值为非零时才对第二个操作数求值；
- 逻辑或(||)，仅当第一个操作数的值为零时，才对第二个操作数求值；
- 条件运算符(?:)始终对第一个操作数求值，然后对第二个或第三个操作数求值；
- “,”运算符，首先对第一个操作数求值，然后对第二个操作数求值。

小贴士：括号的存在可能会更改运算符的应用顺序。 但是，这不影响最低级别操作数的评估顺序，其仍可以按任何顺序进行评估。

遵循 Rule 13.3 和 Rule 13.4 的建议可以避免与表达式评估相关的许多常见的不可预测行为。

示例

此违规示例中，调用 COPY_ELEMENT 宏时，i 被读取两次并被修改两次。 无法确定 i 的操作顺序是否为：

◆ 读取，修改，读取，修改，或

◆ 读取，读取，修改，修改。

```
#define COPY_ELEMENT(index) (a[(index)] = b[(index)])
COPY_ELEMENT(i++);
```

此违规示例中，未指定 v1 和 v2 的读取顺序。

```
extern volatile uint16_t v1, v2;
uint16_t t;

t = v1 + v2;
```

此合规示例中，将读取并修改 PORT。

```
extern volatile uint8_t PORT;

PORT = PORT & 0x80u;
```

如此违规示例中展示的，未指定函数参数的评估顺序，副作用的发生顺序也无法确定。

```
uint16_t i = 0;

/*
 * 无法确定此次调用是否等效于:
 * f(0, 0)
 * 或 f(0, 1)
 */
f(i++, i);
```

函数指示符和函数自变量的求值的相对顺序是未指定的。如这个违规示例中，如果对 g 的调用修改了p，我们无法确定函数指定符p->f 是在g 执行之前还是之后使用 p 的值。

```
p->f(g(&p));
```

参阅

Dir 4.9, Rule 13.1, Rule 13.3, Rule 13.4

Rule 13.3 包含自增(++)或自减(--)运算符的完整表达式，除由自增或自减运算符引起的副作用外，不应有其他潜在的副作用

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15; Undefined 32]

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

展开

函数调用在本规则被认为是一种副作用。(具体见下面示例描述)

完整表达式的所有子表达式都应就本规则的目的进行评估，即使它们在标准 C 中指定不被评估(见 Rule 13.2，在标准 C 中，逻辑比较和条件运算符中，其子表达式并不会全部被求值并评估，下面示例中，针对这种情况也有描述)。

原理

我们不建议将自增和自减运算符与其他运算符混合使用，因为：

- 可能会严重损害代码的可读性；
- 它在语句中引入了其他潜在的不确定性行为(由Rule 13.2 涵盖)。

将这些操作与其他操作符隔离使用会使代码更加清楚，设计意图更加清晰。示

例

下面的表达式违规：

```
u8a = u8b++
```

下面的违规表达式：

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

按下述顺序编写时，其行为会更清晰：

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

以下均为合规示例，因为每个表达式中唯一的副作用是由自增或自减运算符引起的。

```
x++;
a[i]++;
b.x++;
c->x++;
++(*p);
*p++;
(*p)++;
```

以下示例均违规，因为它们同时包含函数调用以及自增或自减运算符：

```
if ((f() + --u8a) == 0u)
{
}

g(u8b++);
```

即使包含自增或自减运算符或某些其他副作用的子表达式不被评估，以下示例依然判定为违规(此即上面“展开”里描述的：子表达式即使在标准 C 中不被评估，也应就此规则进行评定)：

```
u8a = (1u == 1u) ? 0u : u8b++;

if (u8a++ == ((1u == 1u) ? 0u : f()))
{
}
```

参 阅

Rule 13.2

Rule 13.4　不得使用赋值运算符的结果

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15, 18; Undefined 32]

[Koenig 6]

级别　建议

分析　可判定，单一编译单元

适用　C90，C99

展开

即使包含赋值运算符的表达式不被评估，此规则也适用。(见 Rule 13.2)

原理

我们不建议将赋值运算符(简单或复合)与其他算术运算符结合使用，因为：

- 可能会严重损害代码的可读性；
- 它在语句中引入了其他副作用，使得避免 Rule 13.2 涵盖的未定义行为更加困难。

示例

```
x = y;              /* 合规                                          */
a[x] = a[x = y];    /* 违规 - x = y 的值被使用                       */

/*
 * 违规 - bool_var = false 的结果被使用
 * 可能 bool_var == false 才是本意
 */
if (bool_var = false)
{
}

/* 违规 - 即使 bool_var = true 不会被评估                           */
if ((0u == 0u) || (bool_var = true))
{
}
```

```
/* 违规 - x = f() 的结果被使用                                    */
if ((x = f()) != 0)
{
}

/* 违规 - b += c 的结果被使用                                     */
a[b += c] = a[b];

/* 违规 - c = 0 和 b = c = 0 的结果被使用                         */
a = b = c = 0;
```

参 阅

Rule 13.2

Rule 13.5 逻辑与(&&)和逻辑或(||)的右操作数不得含有持久性副作用

级别 必要

分析 不可判定，系统范围

适用 C90，C99

原理

逻辑运算符“&&”和“||”的右侧操作数是否求值取决于左侧操作数的值。如果右侧操作数包含副作用，则可能会发生与程序员期望相反的那些副作用。

如果对右操作数的求值将产生副作用，且该副作用在程序中出现该表达式的点处不是持久的，则无论是否对右操作数求值都没有关系。

术语“持久性副作用”在附录 J 中定义。

示例

```
uint16_t f(uint16_t y)
{
    /* 这些副作用仅调用者可见，所以他们不持久                    */
    uint16_t temp = y;

    temp = y + 0x8080U;

    return temp;
}

uint16_t h(uint16_t y)
{
```

```
    static uint16_t temp = 0;

    /* 此为持久副作用                                    */
    temp = y + temp;

    return temp;
}

void g(void)
{
    /* 合规 - f() 具有非持久副作用                       */
    if (ishigh && (a == f(x)))
    {
    }

    /* 违规 - h() 具有持久副作用                         */
    if (ishigh && (a == h(x)))
    {
    }
}

volatile uint16_t v;
         uint16_t x;

/* 违规 - 对易失性(volatile)对象 v 的访问属于持久副作用 */
if ((x == 0u) || (v == 1u))
{
}

/* 如果fp指向的是具有持久副作用的函数，此处就是违规的    */
(fp != NULL) && (*fp)(0);
```

Rule 13.6 sizeof 运算符的操作数不得包含任何可能产生副作用的表达式

C99 [Unspecified 21]

级别 强制

分析 可判定，单一编译单元

适用 C90，C99

展开

sizeof 运算符的操作数中出现的任何表达式通常都不会被求值。此规则要求对任何此类表达式的评估都不应包含副作用，无论是否实际对其进行评估。

在此规则中，函数调用被认为是副作用。

原理

sizeof 运算符的操作数可以是表达式，也可以某个指定的类型。如果操作数包含一个表达式，则有可能发生了编程错误：当在大多数情况下实际上不对表达式进行求值时，程序员期望该表达式被求值。

C90 标准指出，在运行时不会评估出现在操作数中的表达式。

C99 标准中，通常不在运行时评估在操作数中出现的表达式。但是，如果操作数包含长度可变的数组类型，则将在必要时对数组大小表达式进行求值。如果可以在不评估数组大小表达式的情况下确定结果，则不确定是否评估结果。

例外

在 sizeof(V)表达式中，若 V 是“volatile”修饰的非变长数组类型的左值，这种情况是被认可的。

示例

```
volatile int32_t i;
         int32_t j;
         size_t  s;

s = sizeof(j);          /* 合规         */
s = sizeof(j++);        /* 违规         */
s = sizeof(i);          /* 合规 - 例外  */
s = sizeof(int32_t);    /* 合规         */
```

在此示例中，最终的 sizeof 表达式说明了可变长度数组 size 表达式如何对类型的大小没有影响。操作数的类型为“指向形参为 int32_t 型长度为 v 的数组的函数的 n 个指针的数组”。由于操作数具有可变长度数组类型，因此将对其求值。但是，n 个函数指针的数组大小不受那些函数指针类型的参数列表的影响。因此，是否会评估易失性的合格对象(volatile-qualified object)v 并不确定，并且是否出现其副作用也不确定。

```
volatile uint32_t v;

void f(int32_t n)
{
    size_t s;
    s = sizeof(int32_t[n]);                 /* 合规 */
    s = sizeof(int32_t[n++]);               /* 违规 */
    s = sizeof(void (*[n])(int32_t a[v]));  /* 违规 */
}
```

参 阅

Rule 18.8

8.14 控制语句表达式(Control statement expressions)

本节中的某些规则使用术语“循环计数器”。 循环计数器定义为满足以下条件的对象，数组元素或结构/联合的成员：

1. 它具有标量类型；
2. 其值在给定循环实例的每次循环中单调变化； 和
3. 参与退出循环的判定。

小贴士：第二个条件意味着循环计数器的值必须在循环的每次循环中更改，并且对于给定的循环实例，它必须始终沿相同的方向更改。 但是，它可能在不同的实例上朝不同的方向变化，例如有时向后读取数组的元素，有时向前读取它们。

根据此定义，一个循环不必有且只有一个循环计数器：循环可以没有循环计数器，也可以具有多个循环计数器。 有关 for 循环中对循环计数器的进一步限制，请参见 Rule 14.2。

以下代码片段显示了循环及其相应循环计数器的示例。

在此循环中，i 是循环计数器，因为它具有标量类型，单调变化(增加)，并且涉及循环终止条件。

```
for (uint16_t i = 0; a[i] < 10; ++i)
{
}
```

下面循环没有循环计数器。对象 count 具有标量类型并且单调变化，但不涉及终止条件。

```
extern volatile bool_t b;
uint16_t count = 0;

while (b)
{
    count = count + 1U;
}
```

以下代码中，i 和 sum 均为标量且单调变化(分别减小和增大)。但 sum 并不是循环计数器，因为它不参与退出循环的判定。

```
uint16_t sum = 0;

for (uint16_t i = 10U; i != 0U; --i)
{
    sum += i;
}
```

在下面循环中，p 是循环计数器。 它虽然不涉及循环控制表达式，但涉及通过 break 语句退出循环的判定。

```
extern volatile bool_t b;
extern char *p;

do
{
    if (*p == '\0')
    {
        break;
    }
    ++p;
} while (b);
```

在下面循环示例中，循环计数器为 p->count。

```
struct s
{
    uint16_t count;
    uint16_t a[10];
};

extern struct s *p;

for (p->count = 0U; p->count < 10U; ++(p->count))
{
    p->a[p->count] = 0U;
}
```

Rule 14.1　循环计数器的基本类型不能为浮点型

级别　必要

分析　不可判定，系统范围

适用　C90，C99

原理

使用浮点型数据作为循环计数器时，舍入误差的积累可能会导致预期的循环次数与实际不匹配。当循环步长不是浮点数舍入后所能表示的值的时候，这种情况就有可能发生。

即使带有浮点型循环计数器的循环在一个实现中可能行为正确，但在另一个实现中它可能会得出不同的循环次数。(因为实现的不同，浮点数的舍入结果可能不同)

示例

在下面的违规示例中，循环结束时 counter 的值不太可能为 1000。

```
uint32_t counter = 0u;
```

```
for (float32_t f = 0.0f; f < 1.0f; f += 0.001f)
{
    ++counter;
}
```

下面的合规示例使用整数循环计数器来保证 1000 次循环，并在循环内使用它生成 f。

```
float32_t f;

for (uint32_t counter = 0u; counter < 1000u; ++counter)
{
    f = (float32_t)counter * 0.001f;
}
```

下面的 while 循环违规，因为浮点数 f 被用作循环计数器。

```
float32_t f = 0.0f;

while (f < 1.0f)
{
    f += 0.001f;
}
```

下面的 while 循环合规，因为浮点数 f 未被用作循环计数器。

```
float32_t f;
uint32_t u32a;

f = read_float32();

do
{
    u32a = read_u32();
    /* f 在循环内未被修改，所以不作为循环计数器处理 */
} while (((float32_t)u32a - f) > 10.0f);
```

参 阅

Rule 14.2

Rule 14.2 for 循环应为良好格式

级别 必要

分析 不可判定，系统范围

适用 C90，C99

展开

for 循环语句包含三个子句，此规则对它们的要求为：

第一个子句

- 不得为空，或
- 应为循环计数器分配一个值，或
- 应定义并初始化循环计数器(C99)。

第二个子句

- 应该是没有持久副作用的表达式，并且
- 应使用循环计数器和可选的循环控制标志，并且
- 不得使用在 for 循环主体中修改的任何其他对象。

第三个子句

- 应该是一个表达式，其唯一的持久副作用是修改循环计数器的值，并且
- 不得使用在 for 循环主体中修改的对象。

for 循环中只能有一个循环计数器，且不能在 for 循环主体中修改。

循环控制标志定义为：在第二个子句使用的基本型为布尔型的对象的单个标识符。

for 循环主体的行为包括了在该主体内调用的所有函数的行为。

原理

for 循环语句提供了一种通用循环工具。使用形式受限的循环可使代码更易于查看和分析。

例外

三个子句都可以为空，例如 for (;;)，以便允许无限循环。

示例

在下面的 C99 示例中，i 是循环计数器，flag 是循环控制标志。

```
bool_t flag = false;

for (int16_t i = 0; (i < 5) && !flag; i++)
{
    if (C)
    {
        flag = true;    /* 合规 - 允许提前终止循环                  */
    }

    i = i + 3;          /* 违规 - 循环主体内不允许改变循环计数器    */
}
```

参阅

Rule 14.1, Rule 14.3 , Rule 14.4

Rule 14.3　控制表达式不得是值不变的

级别　必要

分析　不可判定，系统范围

适用　C90，C99

展开

此规则适用于：

- if，while，for，do…while 和 switch 语句的控制表达式；
- ?:运算符的第一个操作数。

原理

如果控制表达式的值不变，则意味着可能存在编程错误。因存在不变表达式而无法到达的任何代码都可以由编译器删除。例如，这可能会导致从可执行文件中删除防御性代码。

例外

1. 允许使用用于创建无限循环的不变的控制表达式。
2. 允许 do…while 循环使用值为 0 的布尔型控制表达式。

示例

```
s8a = (u16a < 0u) ? 0 : 1;          /* 违规 - u16a 永远 >= 0        */

if (u16a <= 0xffffu)
{
                                    /* 违规 - 永远成立              */
}

if (2 > 3)
{
                                    /* 违规 - 永远不成立            */
}

for (s8a = 0; s8a < 130; ++s8a)
{
                                    /* 违规 - 永远成立              */
}

if ((s8a < 10) && (s8a > 20))
{
                                    /* 违规 - 永远不成立            */
```

```
}

if ((s8a < 10) || (s8a > 5))
{
                                    /* 违规 - 永远成立              */
}

while (s8a > 10)
{
    if (s8a > 5)
    {
                                    /* 违规 - s8a 并非 volatile     */
    }
}

while (true)
{
                                    /* 合规 - 例外 1                */
}

do
{
                                    /* 合规 - 例外 2                */
} while (0u == 1u);

const uint8_t numcyl = 4u;

/*
 * 违规 - 编译器可能会假定numcyl始终具有值4, 而这种情况是允许的
 */
if (numcyl == 4u)
{
}

const volatile uint8_t numcyl_cal = 4u;

/*
 * 合规 - 编译器假设numcyl_cal可以通过外部方法更改(例如 汽车校准工具),
 * 即使程序无法修改其值
 */
if (numcyl_cal == 4u)
{
}
```

```
uint16_t n;                          /* 10 <= n <= 100                */
uint16_t sum;

sum = 0;

for (uint16_t i = (n - 6u); i < n; ++i)
{
    sum += i;
}

if ((sum % 2u) == 0u)
{
    /*
     * 违规 - sum是6个连续的非负整数的和，因此必为奇数，if语句的控制表
     * 达式将始终为false
     */
}
```

参阅

Rule 2.1, Rule 14.2

Rule 14.4 if 语句和循环语句的控制表达式的基本类型应为布尔型

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

for 循环语句的控制表达式是可选的。 此规则不要求表达式存在，但如果存在，则要求表达式的基本类型为布尔型。

原理

强类型检查要求 if 语句或迭代语句的控制表达式的基本类型为布尔型。

示例

```
int32_t *p, *q;

while (p)           /* 违规 - p 为指针  */
{
}

while (q != NULL)   /* 合规             */
{
```

```
}

while (true)        /* 合规             */
{
}

extern bool_t flag;

while (flag)        /* 合规             */
{
}

int32_t i;

if (i)              /* 违规             */
{
}

if (i != 0)         /* 合规             */
{
}
```

参阅

Rule 14.2, Rule 20.8

8.15 控制流(Control flow)

Rule 15.1 不应使用 goto 语句

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

原理

无限制地使用goto 可能会导致程序结构混乱且难以理解。

在某些情况下，完全禁止 goto 需要引入标志以确保正确的控制流，但这些标志本身可能不如它们替换的 goto 清晰。因此，如果不遵循此规则，则应在遵循 Rule 15.2 和Rule 15.3 的指导下，有限制的使用 goto。

参阅

Rule 9.1, Rule 15.2, Rule 15.3, Rule 15.4

Rule 15.2 goto 语句仅允许跳到在同一函数中声明的稍后位置的标签

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

无限制地使用goto 可能会导致程序结构混乱且难以理解。

限制 goto 的使用以禁止“向回”跳转，可确保仅在使用该语言提供的循环语句时才发生迭代，从而有助于最大程度地减少可视代码的复杂性。

示例

```
void f(void)
{
    int32_t j = 0;
L1:
    ++j;

    if (10 == j)
    {
        goto L2;    /* 合规 */
    }

    goto L1;        /* 违规 */

L2 :
    ++j;
}
```

参阅

Rule 15.1, Rule 15.3, Rule 15.4

Rule 15.3 goto 语句引用的标签必须在 goto 语句所在代码块或包含该代码块的上级代码块中声明

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

为符合此规则的设计目的，将不包含复合语句(复合语句：{}括起来的语句)的 switch 子句也视为一个代码块(即不用{}括起来的 case 子句也被视为独立的代码块)。

原理

无限制地使用goto 可能会导致程序结构混乱且难以理解。

阻止代码块之间或嵌套代码块之间的跳转有助于最大程度地减少可视代码的复杂性。

小贴士：在 C99 中，使用可变修改类型时，限制更大。 试图从具有可变修改类型的标识符的范围外跳入这样的范围会导致约束冲突(即编译错误)。

示例

```
void f1(int32_t a)
{
    if (a <= 0)
    {
        goto L2;    /* 违规 - L2 在其他代码块中                  */
    }

    goto L1;        /* 合规 - L1 在同代码块                      */

    if (a == 0)
    {
        goto L1;    /* 合规 - L1 在包含此goto语句的上级代码块中 */
    }

    goto L2;        /* 违规 - L2 在子代码块中                    */
L1:
    if (a > 0)
    {
    L2:
        ;
    }
}
```

在下面的示例中，标签 L1 在包含 goto 语句的块的块中定义。但是，发生了从一个swtich 子句到另一个 switch 子句的跳转，而由于此规则的目的，将 switch 的子句视为代码块，因此这个 goto 语句是违规的。

```
switch (x)
{
    case 0:
        if (x == y)
        {
            goto L1;
        }
        break;
    case 1:
        y = x;
```

```
    L1:
        ++x;
        break;
    default:
        break;
}
```

参阅

Rule 9.1, Rule 15.1, Rule 15.2, Rule 15.4, Rule 16.1

Rule 15.4　最多只能有一个用于终止循环语句的 break 或goto 语句

级别　建议

分析　可判定，单一编译单元

适用　C90，C99

原理

限制循环的退出数量有助于最大程度地减少可视代码的复杂性。仅使用一个break 或 goto 语句可以在需要提前终止循环时创建单个辅助出口路径。

示例

下面嵌套的两个循环均合规，因为每个循环都只有一个 break 语句用于提前终止循环。

```
for (x = 0; x < LIMIT; ++x)
{
    if (ExitNow(x))
    {
        break;
    }

    for (y = 0; y < x; ++y)
    {
        if (ExitNow(LIMIT - y))
        {
            break;
        }
    }
}
```

下面循环违规，因为有多个 break 和 goto 语句用于提前终止循环。

```
for ( x = 0; x < LIMIT; ++x )
{
    if (BreakNow(x))
    {
```

```
        break;
    }
    else if (GotoNow(x))
    {
        goto EXIT;
    }
    else
    {
        KeepGoing (x);
    }
}

EXIT:
    ;
```

在下面示例中，内部 while 循环是合规的，因为它只有一个 goto 语句可能导致其提前终止。 但是，外部 while 循环是违规的，因为它可以通过 break 语句或内部 while 循环中的 goto 语句提前终止。

```
while (x != 0u)
{
    x = calc_new_x();

    if (x == 1u)
    {
        break;
    }

    while (y != 0u)
    {
        y = calc_new_y();

        if (y == 1u)
        {
            goto L1;
        }
    }
}

L1:
    z = x + y;
```

参阅

Rule 15.1, Rule 15.2, Rule 15.3

Rule 15.5　应仅在函数的末尾有单个函数出口

[IEC 61508-3 Table B.9], [ISO 26262-6 Table 8]

级别　建议

分析　可判定，单一编译单元

适用　C90，C99

展开

一个函数最多只能有一个 return 语句。

使用 return 语句时，它应该是复合语句构成的函数主体的最后一条语句。

原理

IEC 61508 和 ISO 26262 都要求函数有单出口，这是模块化方法要求的一部分。

提早返回可能会导致意外删除函数终止代码。

如果函数的出口点散布着会产生持久副作用的语句，则很难确定执行函数时将发生哪些副作用。

示例

在下面的违规代码示例中，提前返回被用于验证函数形参。

```
bool_t f(uint16_t n, char *p)
{
    if (n > MAX)
    {
        return false;
    }

    if (p == NULL)
    {
        return false;
    }

    return true;
}
```

参 阅

Rule 17.4

Rule 15.6　循环语句和选择语句的主体应为复合语句

[Koenig 24]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

循环语句(while，do…while 或 for)或选择语句(if，else,switch)的主体应为复合语句。

复合语句：{}括起来的语句。

原理

开发人员可能会错误地认为，语句的序列由于缩进而形成了循环语句或选择语句的主体。 在控制表达式之后意外出现的分号是一种特殊的危险，它将导致控制声明为空。 使用复合语句就可以避免上述问题，并清楚地定义哪些语句实际上构成了主体。

此外，缩进有可能导致开发人员将 else 语句与错误的 if 关联。

例外

如果 if 语句紧随 else 出现，则这个 if 语句不必被包含在符合语句中(即允许 else if 语句出现)。

示例

复合语句的布局及包围它的花括号位置是样式问题，本文档不解决。 以下示例中使用的样式不是强制性的。

维护以下代码：

```
while (data_available)
    process_data();         /* 违规 */
```

可能会错误的写成

```
while (data_available)
    process_data();         /* 违规 */
    service_watchdog();
```

而本意是将 service_watchdog()添加到循环主体中。 复合语句的使用可以大大减少这种情况的发生。

下面示例中，action_2()看上去是第一个 if 的 else 语句。

```
if (flag_1)
    if (flag_2)             /* 违规 */
        action_1();         /* 违规 */
else
    action_2();             /* 违规 */
```

而它的实际行为是：

```
if (flag_1)
{
    if (flag_2)
    {
        action_1();
    }
    else
    {
        action_2();
    }
}
```

复合语句的使用可确保明确定义 if 和else 间的关联。

例外允许使用else if，如下所示

```
if (flag_1)
{
    action_1();
}
else if (flag_2) /* 合规 - 符合例外 */
{
    action_2();
}
else
{
    ;
}
```

以下示例显示了虚假的分号如何导致错误：

```
while (flag);       /* 违规 */
{
    flag = fn();
}
```

以下示例显示了循环主体为空的合规方法：

```
while (!data_available)
{
}
```

Rule 15.7　所有的 if…else if 构造都应以 else 语句结束

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

当 if 语句后接一个或多个 else if 构造的序列时，应始终以 else 语句作为结尾。 else 语句至少应包含一个副作用或注释。

原理

用 else 语句终止 if…else if 序列的序列是防御性编程，相似的，还有 switch 语句中对 default 子句的要求(请参见规则 16.5)。

else 语句必须具有副作用或注释，以确保对所需的行为给出肯定的指示，以帮助代码检查过程。

小贴士：简单的 if 语句，不需要 else 语句作为结尾。

简单的 if 语句：没有 else if 分支，且语句主体很短的 if 语句。

建议：简单的累加器这样的无 else if 分支，且语句主体中仅有一条语句的 if 语句，可不以 else 语句结束；语句主体中语句多于两条的，建议仍增加含有注释的 else 语句作为结束，以明确设计意图。

示例

以下违规示例中，没有明确使用 else 语句结尾以指示不再执行任何操作。

```
if (flag_1)
{
    action_1();
}
else if (flag_2)
{
    action_2();
}
/* 违规 */
```

下面显示了一个符合标准的终止 eles 示例。

```
else
{
    ; /* 无任何必要行为 - ";"是可选的(即：重要的是要有这段注释) */
}
```

参 阅

Rule 16.5

8.16 Switch 语句(Switch statements)

Rule 16.1 Switch 语句应格式正确

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

如果 switch 语句符合以下语法规则指定的 C switch 语句的子集，则应认为它是格式正确的。 如果此处语法规则的名称与“标准”中定义的语法名称相同，则应用它代替标准版本中的 switch 语句语法。 否则，《标准》中给出的所有语法规则均保持不变。

```
switch-statement:
    switch (switch-expression) { case-label-clause-list final-default-clause-list }
    switch (switch-expression) { initial-default-clause-list case-label-clause-list }

case-label-clause-list:
    case-clause-list
    case-label-clause-list case-clause-list

case-clause-list:
    case-label switch-clause
    case-label case-clause-list

case-label:
    case constant-expression:

final-default-clause-list:
    default:  switch-clause
    case-label final-default-clause-list

initial-default-clause-list:
    default:  switch-clause
    default:  case-clause-list

switch-clause:
    statement-list_opt break;
    C90: { declaration-list_opt statement-listopt break; }
    C99: { block-item-list_opt break; }
```

除非此语法明确允许，否则 case 和 default 关键字不应出现在 switch 语句主体内的任何位置。

小贴士：此规则对 switch 语句施加的一些限制在“参阅”部分中引用的规则中进行了阐述。 因此，

代码可能同时违反此规则和更具体的规则之一。

小贴士：*术语 switch 标签(switch label)在特定switch 语句规则的文本中使用，表示case(case label)标签或 default 标签(default label)。*

原理

C 语言中 switch 语句的语法不是特别严格，并且可以允许复杂的，非结构化的行为。 此规则和其他规则在 switch 语句上强加了一个简单且一致的结构限制。

示例

本节中其他规则的示例与此规则也相关。

参阅

Rule 15.3, Rule 16.2, Rule 16.3, Rule 16.4, Rule 16.5, Rule 16.6

Rule 16.2　switch 标签只能出现在构成 switch 语句主体的复合语句的最外层

此处并未照直翻译，而是采用了原理中的描述

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

C 标准允许将 switch 标签(即 case 标签或 default 标签)放置在 switch 语句主体中的任何语句之前，这可能会导致结构混乱的代码。 为了防止这种情况，switch 标签只能出现在构成 switch 语句主体的复合语句的最外层。

示例

```
switch (x)
{
    case 1:             /* 合规 */
        if (flag)
        {
    case 2:             /* 违规 */
            x = 1;
        }
        break;
    default:
        break;
}
```

参阅

Rule 16.1

Rule 16.3 每一个 switch 子句(switch-clause)都应以无条件 break 语句终止

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

如果开发人员未能以 break 语句结束 switch 子句，则控制流将“落入”其下面的 switch 子句，或者，如果没有这样的子句，则控制流将退出并进入 switch 语句后的语句。 尽管有时会故意设计进入随后的 switch 子句中，但这通常是一个错误。 在 switch 语句末尾出现的未终止的 switch 子句可能会落入以后添加的任何 switch 子句中。

为了确保可以检测到此类错误，每个 switch 子句中的最后一条语句都应为 break 语句，如果 switch 子句为复合语句，则该复合语句中的最后一条语句应为 break 语句。

小贴士：一个 switch 子句(switch-clause)被定义为包含至少一个语句。此规则允许出现两个连续的标签(case 或 default)没有任何中间语句的情况。

示例

```
switch ( x )
{
    case 0:
        break;          /* 合规 - 无条件break                    */
    case 1:             /* 合规 - 允许无中间语句的连续标签       */
    case 2:
        break;          /* 合规                                  */
    case 4:
        a = b;          /* 违规 - break被省略                    */
    case 5:
        if ( a == b )
        {
            ++a;
            break;      /* 违规 - 有条件break                    */
        }
    default:
        ;               /* 违规 - default 也必须有 break         */
}
```

参阅

Rule 16.1

Rule 16.4 每个 switch 语句都应具有 default 标签

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

在最后的 break 语句之前，default 标签后面的 switch 子句应至少包含以下任一内容：

- 一条语句，或
- 一条注释。

原理

对 default 标签的要求的目的是防御性编程。Default 标签后面的所有语句均旨在采取适当的措施。如果标签后没有语句，则可以使用注释来解释为什么未采取任何具体措施。

示例

```
int16_t x;

switch (x)
{
    case 0:
        ++x;
        break;
    case 1:
    case 2:
        break;
                        /* 违规 - 缺少 default 标签                */
}

int16_t x;

switch (x)
{
    case 0:
        ++x;
        break;
    case 1:
    case 2:
        break;
    default:            /* 合规 - 存在 default 标签                */
        errorflag = 1;  /* 若可能，此处应非空                      */
        break;
```

```
}

enum Colours { RED, GREEN, BLUE } colour;

switch (colour)
{
    case RED:
        next = GREEN;
        break;
    case GREEN:
        next = BLUE;
        break;
    case BLUE:
        next = RED;
        break;
                        /* 违规 - 缺少 default 标签.
                         * 即使对所有枚举值都有处理，也无法保证colour
                         * 采用了这些值之一 */
}
```

参阅

Rule 2.1, Rule 16.1

Rule 16.5 Default 标签应作为 switch 语句的第一个或最后一个 switch 标签

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

通过此规则，可以轻松在 switch 语句中找到 default 标签。

示例

```
switch (x)
{
    default:    /* 合规 - default 作为第一个标签          */
        case 0:
        ++x;
        break;
    case 1:
    case 2:
        break;
}
```

```
switch (x)
{
    case 0:
        ++x;
        break;
    default:    /* 违规 - default 标签与 case 标签混杂  */
        x = 0;
        break;
    case 1:
    case 2:
        break;
}

switch (x)
{
    case 0:
        ++x;
        break;
    case 1:
    case 2:
        break;
    default:    /* 合规 - default 作为最后一个标签      */
        x = 0;
        break;
}
```

参阅

Rule 15.7, Rule 16.1

Rule 16.6　每个 switch 语句应至少有两个 switch 子句

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

仅有单个路径的 switch 语句是多余的，且其存在很可能表示出现了编程错误。

示例

```
witch (x)
{
    default:    /* 违规 - switch 语句多余   */
```

```
        x = 0;
        break;
}
switch (y)
{
    case 1:
    default:    /* 违规 - switch 语句多余    */
        y = 0;
        break;
}
switch (z)
{
    case 1:
        z = 2;
        break;
    default:    /* 合规                      */
        z = 0;
        break;
}
```

参 阅

Rule 16.1

Rule 16.7 switch 语句的控制表达式(switch-expression)的基本类型不得是布尔型

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

标准 C 要求 switch 语句的控制表达式具有整数类型。由于(C 语言中)布尔值是用整数实现的，因此可能会出现由布尔表达式控制的 switch 语句。在这种情况，用 if-else 构造会更合适。

示例

```
switch (x == 0)     /* 违规 - 布尔表达式                */
{                   /* 这种情况下, "if-else" 更具逻辑性 */
    case false:
        y = x;
        break;
    default:
        y = z;
        break;
}
```

8.17 函数(Functions)

Rule 17.1 不得使用<stdarg.h>的功能

C90 [Undefined 45, 70－76], C99 [Undefined 81, 128－135]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

扩展

函数 va_list，va_arg，va_start，va_end 和C99 中的 va_ copy 均禁止使用。

原理

标准 C 中列出了许多与<stdarg.h>功能相关的不确定行为的实例，包括：

- 在使用 va_start 的函数的末尾未使用 va_end；
- va_arg 在同一 va_list 上的不同函数中使用；
- 参数的类型与 va_arg 指定的类型不兼容。

示例

```
#include <stdarg.h>

void h(va_list ap)              /* 违规                          */
{
    double y;

    y = va_arg(ap, double);     /* 违规                          */
}

void f(uint16_t n, ...)
{
    uint32_t x;

    va_list ap;                 /* 违规                          */

    va_start(ap, n);            /* 违规                          */
    x = va_arg(ap, uint32_t);   /* 违规                          */

    h(ap);
```

```
    /* 未定义 - ap是不确定的，因为h()中使用了va_arg                */
    x = va_arg(ap, uint32_t);    /* 违规                          */

    /* 未定义 - 函数返回前，未使用 va_end()                        */
}

void g(void)
{
    /* 未定义 - uint32_t: 当 f 使用 va_arg() 时，双精度不匹配    */
    f(1, 2.0, 3.0);
}
```

Rule 17.2 函数不得直接或间接调用自身(不得使用递归函数)

级别 必要

分析 不可判定，系统范围

适用 C90，C99

原理

递归会带来堆栈空间溢出的危险，这可能会导致严重的故障。除非对递归进行了严格控制，否则无法在执行之前确定最坏情况的堆栈使用情况。

Rule 17.3 禁止隐式声明函数

C90 [Undefined 6, 22, 23]

级别 强制

分析 可判定，单一编译单元

适用 C90

原理

在原型存在的情况下进行函数调用，(C 语言编译器的)约束条件可确保参数的数量与参数的数量匹配，并且可以将每个参数分配给其相应的参数。

如果隐式声明函数，C90 编译器将假定该函数的返回类型为 int。由于隐式函数声明不提供原型，因此编译器将没有有关函数参数的数量及其类型的信息。不正确的类型转换可能导致传递参数和分配返回值以及其他未定义的行为。

示例

如果函数 power 声明为：

```
extern double power(double d, int n);
```

但是该声明在以下代码中不可见，则将发生未定义的行为。

```
void func(void)
{
    /* 违规 - 返回类型和两个形参类型都不正确     */
    double sq1 = power( 1, 2.0);
}
```

参阅

Rule 8.2, Rule 8.4

Rule 17.4　具有非 void 返回类型的函数的所有退出路径都应为具有带有表达式的显式 return 语句

C90 [Undefined 43], C99 [Undefined 82]

级别　强制

分析　可判定，单一编译单元

适用　C90，C99

原理

函数的返回值有 return 语句的表达式提供。如果非 void 函数没有返回值，但是调用函数使用了返回的值，则该行为是不确定的。为避免这种情况，限制非 void 函数：

- 每个 return 语句都必须有一个表达式，并且
- 如果不遇到 return 语句，禁止执行到函数的末尾。

小贴士：C99 强制约束了非 void 函数中的每个 return 语句都要有表达式作返回值。

示例

```
int32_t absolute(int32_t v)
{
    if (v < 0)
    {
        return v;
    }

    /* 违规 - 可以在无return语句的情况下到达函数的末尾  */
}

uint16_t lookup(uint16_t v)
{
    if ((v < V_MIN) || (v > V_MAX))
    {
```

```
        /* 违规 - 无返回值，C99约束                    */
        return;
    }
    return table[v];
}
```

参 阅

Rule 15.5

Rule 17.5 与数组型函数形参对应的函数入参应具有适当数量的元素

级别 建议

分析 不可判定，系统范围

适用 C90，C99

展开

如果将形参声明为具有指定大小的数组，则每次函数调用时的相应参数应指向一个对象，且该对象至少具有与数组一样多的元素。

原理

将函数形参声明为数组比声明为指针能更清楚地指定函数接口。(声明为数组时，)该函数期望的最小元素数已明确说明，而使用指针则不能。

函数形参声明为不指定大小的数组表示该函数可以处理任何大小的数组。在这种情况下，预计将通过其他方式传达数组大小，例如，通过将数组长度作为另一个参数进行传递，或通过使用定值作为数组终止(最大下标值)。

使用数组边界是一种建议方法，因为它允许在函数体内实现越界检查，并在参数传递上进行额外检查。在 C 语言中，将大小错误的数组传递给具有指定大小的参数是合法的，这可能会导致意外行为。

示例

```
/*
 * 该函数入参输入不会超过范围
 * array1[0] ... array1[3]
 */
void fn1(int32_t array1[4]);

/* 该函数可以处理任意大小的数组                    */
void fn2(int32_t array2[]);

void fn(int32_t *ptr)
{
```

```
    int32_t arr3[3] = { 1, 2, 3 };
    int32_t arr4[4] = { 0, 1, 2, 3 };

    /* 合规 - 数组大小与原型匹配                */
    fn1(arr4);

    /* 违规 - 数组大小与原型不匹配              */
    fn1(arr3);

    /* 合规 - 仅当ptr指向的数组元素数至少为4时  */
    fn1(ptr);

    /* 合规                                     */
    fn2(arr4);

    /* 合规                                     */
    fn2(ptr);
}
```

参 阅

Rule 17.6

Rule 17.6　数组形参的声明不得在[]之间包含 static 关键字

C99 [Undefined 71]

级别　强制

分析　可判定，单一编译单元

适用　C99

原理

C99 语言标准为程序员提供了一种机制，可以通知编译器数组形参包含指定的最小元素数。 一些编译器能够利用此信息为某些类型的处理器生成更有效的代码。

如果未能遵循程序员设定，并且元素数少于指定的最小值，则其行为是未指定的。

但典型的嵌入式应用程序中使用的处理器不太可能提供必要的功能以利用程序员提供的附加信息。 程序无法满足预定的最小元素数量的风险超过了任何潜在的性能提升。

示例

下面示例列举了在没有使用符合此规则的 C99 语言功能的情况下，某些引起不确定行为的(在数组形参声明的[]中使用 static 的)使用场景。

```
/* 违规 - 在数组型形参的声明中使用 static */
```

```
uint16_t total(uint16_t n, uint16_t a[static 20])
{
    uint16_t i;

    uint16_t sum = 0U;

    /* 若数组a的实际元素数少于20，则行为未定义 */
    for (i = 0U; i < n; ++i)
    {
        sum = sum + a[i];
    }

    return sum;
}

extern uint16_t v1[10];
extern uint16_t v2[20];

void g(void)
{
    uint16_t x;

    x = total(10U, v1); /* 未定义 - v1 仅有10个元素，少于需要的20个          */

    x = total(20U, v2); /* 符合预期，结果可确定，但这种定义方式本身是违规的 */
}
```

参 阅

Rule 17.5

Rule 17.7　非 void 返回类型的函数的返回值应该被使用

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

C 语言允许在不使用其返回值的情况下调用函数，这可能是错误的(与编程预期不符的)。 如果明确打算不使用函数的返回值，则应将其强制转换为 void 类型。 这具有在不违反Rule 2.2 的情况下使用该值的效果。

示例

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

uint16_t x;

void discarded(uint16_t para2)
{
    func(para2);         /* 违规 - 返回值被(隐式)丢弃     */
    (void)func(para2);   /* 合规                          */
    x = func(para2);     /* 合规                          */
}
```

参阅

Dir 4.7, Rule 2.2

Rule 17.8　不应更改函数形参

级别　建议

分析　不可判定，系统范围

适用　C90，C99

原理

函数形参的行为与具有自动存储持续时间的对象(临时变量)的行为相同。虽然 C 语言允许修改形参，但这样的使用可能会造成混乱并与程序期望值产生冲突。将形参复制到自动对象并修改该副本可能会减少混乱。使用现代编译器，(这种做法)通常不会导致任何存储或执行时间损失。

不熟悉 C 但习惯于其他语言的程序员可能会认为修改一个形参的效果会在调用函数后体现在入参上。**示例**

```
int16_t glob = 0;

void proc(int16_t para)
{
    para = glob; /* 违规 */
}

void f(char *p, char *q)
{
    p = q;       /* 违规 */
    *p = *q;     /* 合规 */
}
```

8.18 指针和数组(Pointers and arrays)

Rule 18.1 指针操作数的算术运算应仅用于寻址与该指针操作数相同数组的元素

C90 [Undefined 30], C99 [Undefined 43, 44, 46, 59]

级别 必要

分析 不可判定，系统范围

适用 C90，C99

展开

C 语言标准中允许创建一个指向数组末尾再加 1 的指针，此规则也允许这么做。但是，反引用(dereferencing, 即*pointer 操作)指向数组末尾再加 1 的指针的行为是未定义的，因此被此规则禁止。

此规则适用于下述所有形式的数组索引：

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

小贴士： *子数组也是数组。*

小贴士： *出于指针算术的目的，C 标准将不属于数组成员的对象视为具有单个元素的数组(C90 第 6.3.6 节，C99 第6.5.6 节)。*

原理

尽管某些编译器可能在编译时就能检查到已超出数组边界，但通常在运行时不会检查无效的数组下标。而无效数组下标的使用会导致程序的错误行为。

运行时派生的数组下标值是最受关注的，因为它们不容易通过静态分析或手动检查进行检查。在可能和可行的情况下，应提供防御性编程性质的代码，以将这些下标值与有效值进行比较，并在必要时采取适当的措施。

如果通过上述表达式获得的结果既不是指向与指针表达式(pointer_expression)所指向的相同数组元素的指针，也不是指向该数组末尾再加 1 的指针，则其行为是未定义的。

多重数组，即“由数组组成的数组”。此规则不允许使用指针算数的方式来寻址不同的子数组。数组小标不得越过其“内部”边界，因为其行为是未定义的。

示例

下面的示例中“+”运算符的使用，也违反了 Rule 18.4.

```
int32_t f1(int32_t * const a1, int32_t a2[10])
{
    int32_t *p = &a1[3];        /* 合规/违规 - 取决于a1                    */

    return *(a2 + 9);           /* 合规                                    */
}

void f2(void)
{
    int32_t data = 0;
    int32_t b =  0;
    int32_t c[10] = { 0 };
    int32_t d[5][2] = { 0 };    /* 二维数组                                */
    int32_t *p1 = &c[0];        /* 合规                                    */
    int32_t *p2 = &c[10];       /* 合规 - 指针越界1                        */
    int32_t *p3 = &c[11];       /* 违规 - 未定义，指针越界多于1            */

    data = *p2;                 /* 违规 - 未定义，反引用越界1的值          */

    data = f1(&b, c);
    data = f1(c, c);

    p1++;                       /* 合规                                    */
    c[-1] = 0;                  /* 违规 - 未定义，超出数组范围             */

    data = c[10];               /* 违规 - 未定义，反引用越界1的值          */

    data = *(&data + 0);        /* 合规 - C语言将data识别为长度为1的数组   */

    d[3][1] = 0;                /* 合规                                    */
    data = *(*(d + 3) + 1);     /* 合规                                    */
    data = d[2][3];             /* 违规 - 未定义，超出内部边界             */

    p1 = d[1];                  /* 合规                                    */
    data = p1[1];               /* 合规 - p1 寻址长度为2的数组             */
}
```

以下示例说明了应用于结构成员的指针算法。由于每个成员本身就是一个对象，因此该规则禁止使用

```
struct
{
    uint16_t x;
    uint16_t y;
    uint16_t z;
    uint16_t a[10];
} s;

uint16_t *p;

void f3(void)
{
    p = &s.x;
    ++p;            /* 合规 - p 指向数组s.x(单元素数组)越界1        */
    p[0] = 1;       /* 违规 - 未定义，反引用越 s.x 界1的指针，但它
                     * 不一定与 s.y 相同                             */
    p[1] = 2;       /* 违规 - 未定义                                 */

    p = &s.a[0];    /* 合规 - p 指向 s.a                             */
    p = p + 8;      /* 合规 - p 仍然指向 s.a                         */
    p = p + 3;      /* 违规 - 未定义，p 指向越 s.a 界1               */
}
```

指针算法从一个成员移动到下一个成员。但是，如果结果指针保持在成员对象的范围之内，它就不会阻止对成员指针的算术运算。

参阅

Dir 4.1, Rule 18.4

Rule 18.2　指针之间的减法应仅用于寻址同一数组元素的指针

C90 [Undefined 31], C99 [Undefined 45]

级别　必要

分析　不可判定，系统范围

适用　C90，C99

原理

此规则适用于以下形式的表达式：

```
pointer_expression_1 - pointer_expression_2
```

如果 pointer_expression_1 和 pointer_expression_2 未指向同一数组的元素，也未指向该数组末尾的元

素，则这是未定义的行为。

示例

```
#include <stddef.h>

void f1(int32_t *ptr)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = &a1[1];
    int32_t *p2 = &a2[10];
    ptrdiff_t diff;

    diff = p1 - a1; /* 合 规 */
    diff = p2 - a2; /* 合 规 */
    diff = p1 - p2; /* 违 规 */
    diff = ptr - p1; /* 违规 */
}
```

参阅

Dir 4.1, Rule 18.4

Rule 18.3　关系运算符>，> =，<和<=不得应用于指针类型的对象，除非它们指向同一对象

C90 [Undefined 33], C99 [Undefined 50]

级别　必要

分析　不可判定，系统范围

适用　C90，C99

原理

如果两个指针未指向同一对象，则尝试在两个指针之间进行比较将产生不确定的行为。

小贴士：*允许寻址数组末尾以外的下一个元素，但不允许访问该元素。*

示例

```
void f1(void)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = a1;

    if (p1 < a1)    /* 合规 */
    {
    }

    if (p1 < a2)    /* 违规 */
```

```
    {
    }
}
struct limits
{
    int32_t lwb;
    int32_t upb;
};
void f2(void)
{
    struct limits limits_1 = { 2, 5 };
    struct limits limits_2 = { 10, 5 };

    if (&limits_1.lwb <= &limits_1.upb) /* 合规 */
    {
    }

    if (&limits_1.lwb > &limits_2.upb)  /* 违规 */
    {
    }
}
```

参 阅

Dir 4.1

Rule 18.4 +，-，+=和-=运算符不得应用于指针类型的表达式

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

原理

使用数组下标语法ptr[expr]进行数组索引是指针算术的首选形式，因为它通常比指针操作更清晰，因此出错率更低。任何显式计算的指针值都有可能访问意外或无效的内存地址。使用数组索引也可以实现这种行为，但是下标语法可以简化手动检查的任务。

C 语言中的指针算法可能会使新手感到困惑。表达式 ptr + 1 可能会错误地解释为ptr 中保存的地址加1。实际上，新的内存地址取决于指针目标的字节大小。如果错误地应用了 sizeof，这种误解可能导致意外的行为。

但是，在某些情况下谨慎地使用++进行指针操作可能会更自然。例如，在内存测试期间顺序访问位置，将内存空间视为位置的连续集合更为方便，并且可以在编译时确定地址范围。

例外

依据 Rule 18.2，指针间的减法是被允许的。

示例

```
void fn1(void)
{
    uint8_t a[10];
    uint8_t *ptr;
    uint8_t index = 0U;

    index = index + 1U; /* 合规 - 此规则仅适用于指针        */

    a[index] = 0U;      /* 合规                             */
    ptr = &a[5];        /* 合规                             */

    ptr = a;
    ptr++;              /* 合规 - ++ 运算符不等于 + 运算符  */
    *(ptr + 5) = 0U;    /* 违规                             */
    ptr[5] = 0U;        /* 合规                             */
}

void fn2(void)
{
    uint8_t array_2_2[2][2] = { { 1U, 2U }, { 4U, 5U } };
    uint8_t i = 0U;
    uint8_t j = 0U;
    uint8_t sum = 0U;

    for(i = 0U; i < 2U; i++)
    {
        uint8_t *row = array_2_2[i];

        for(j = 0U; j < 2U; j++)
        {
            sum += row[j];  /* 合规 */
        }
    }
}
```

在以下示例中，如果 p1 不指向具有至少六个元素的数组，p2 不指向具有至少四个元素的数组，也可能违反 Rule 18.1。

```
void fn3(uint8_t *p1, uint8_t p2[])
{
    p1++;                   /* 合规 */
```

```
    p1 = p1 + 5;            /* 违规 */
    p1[5] = 0U;             /* 合规 */

    p2++;                   /* 合规 */
    p2 = p2 + 3;            /* 违规 */
    p2[3] = 0U;             /* 合规 */
}

uint8_t a1[16];
uint8_t a2[16];
uint8_t data = 0U;

void fn4(void)
{
    fn3(a1, a2);
    fn3(&data, &a2[4]);
}
```

参阅

Rule 18.1, Rule 18.2

Rule 18.5 声明中最多包含两层指针嵌套

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

扩展

一个类型最多允许连续使用两个指针声明符。声明中出现的任何 typedef-name 都将被识别为其所表示的类型。

原理

使用多于层的指针嵌套会严重损害理解代码行为的能力，因此应避免使用。

示例

```
typedef int8_t *INTPTR;

void function(int8_t ** arrPar[])   /* 违规 */
{
    int8_t **  obj2;                /* 合规 */
    int8_t *** obj3;                /* 违规 */
    INTPTR *   obj4;                /* 合规 */
    INTPTR *   const * const obj5;  /* 违规 */
```

```
    int8_t **  arr[10];              /* 合规 */
    int8_t **  (  *parr)[10];        /* 合规 */
    int8_t *   (**pparr)[10];        /* 合规 */
}
struct s
{
    int8_t *   s1;                   /* 合规 */
    int8_t **  s2;                   /* 合规 */
    int8_t *** s3;                   /* 违规 */
};

struct s *   ps1;                    /* 合规 */
struct s **  ps2;                    /* 合规 */
struct s *** ps3;                    /* 违规 */

int8_t ** (  *pfunc1)(void);         /* 合规 */
int8_t ** ( **pfunc2)(void);         /* 合规 */
int8_t ** (***pfunc3)(void);         /* 违规 */
int8_t ***( **pfunc4)(void);         /* 违规 */
```

小贴士：

- *arrPar 的类型为指向 int8_t 的指针的指针，因为用数组类型声明的形参被转换为指向数组的初始元素的指针 - 这是三层指针，故违规。*
- *arr 是指向 int8_t 的指针的指针的数组类型 - 这是合规的；*
- *parr 的类型为：指向(指向(int8_t 的指针)的数组的指针)的指针 - 合规；*
- *pparr 的类型为：指向(指向(int8_t 的指针)的指针的数组)的指针 - 合规。*

Rule 18.6 具有自动存储功能的对象的地址不得复制给在它的生命周期结束后仍会存在的另一个对象

C90 [Undefined 9, 26], C99 [Undefined 8, 9, 40]

级别 必要

分析 不可判定，系统范围

适用 C90，C99

扩展

对象的地址可以通过以下方式复制：

- 赋值；
- 内存移动或复制功能。

原理

当对象的生命周期结束时，该对象的地址将变得不确定。任何不确定地址的使用都会导致不确定的行为。

示例

```
int8_t *func(void)
{
    int8_t local_auto;
    return &local_auto; /* 违规 - &local_auto 在 func返回后，不再确定 */
}
```

在下面的示例中，函数 g 存储其指针形参 p 的副本。如果 p 始终指向具有静态存储持续时间的对象，则代码符合此规则。但是，在给出的示例中，p 确实指向具有自动存储持续时间的对象。在这种情况下，复制参数 p 是违规的。

```
uint16_t *sp;

void g(uint16_t *p)
{
    sp = p;          /* 违规 - f的形参u的地址复制到静态指针sp     */
}

void f(uint16_t u)
{
    g(&u);
}

void h(void)
{
    static uint16_t *q;

    uint16_t x = 0u;

    q = &x;              /* 违规 - &x 存储在具有更长寿命的对象中 */
}
```

Rule 18.7　不得声明灵活数组成员

C90 [Undefined 9, 26], C99 [Undefined 8, 9, 40]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

原理

灵活的数组成员最有可能与 Dir 4.12 和 Rule 21.3 禁止的动态内存分配结合使用。

灵活数组成员的存在可能会以程序员无法期望的方式修改 sizeof 运算符的行为。将包含灵活数组成员的结构分配给相同类型的另一个结构可能无法按预期的方式执行，因为它所复制的那些元素，只到灵活数组成员的开头，却不包括灵活数组。

示例

```
#include <stdlib.h>

struct s
{
    uint16_t len;
    uint32_t data[];    /* 违规 - 灵活数组成员  */
} str;

struct s *copy(struct s *s1)
{
    struct s *s2;

    /* 此处为了简洁，省略了malloc()返回值检查   */
    s2 = malloc(sizeof(struct s) + (s1->len * sizeof(uint32_t)));

    *s2 = *s1;          /* 只复制了 s1->len     */

    return s2;
}
```

参阅

Dir 4.12, Rule 21.3

Rule 18.8　不得使用可变长数组类型

C99 [Unspecified 21; Undefined 69, 70]

级别　必要

分析　可判定，单一编译单元

适用　C99

原理

当在块或函数原型中声明的数组的大小不是整数常量表达式时，将指定可变长度数组类型。它们通常被实现为存储在堆栈中的可变大小的对象。因此，使用它们可能无法静态确定必须为堆栈保留的内存量。

如果可变长度数组的大小为负或零，则行为是未定义的。

如果在要求与另一个数组类型(可能本身是可变长度)兼容的上下文中使用了可变长度数组，则数组类型的大小应相同。 此外，所有大小均应为正整数。 如果不满足这些要求，则行为是不确定的。

如果在 sizeof 运算符的操作数中使用了可变长度数组类型，则在某些情况下，不确定是否能正确评估数组大小表达式。

可变长度数组类型的每个实例在其生命周期的开始都具有固定的大小。这可能会引起令人困惑的行为，例如：

```
void f(void)
{
    uint16_t n = 5;

    typedef uint16_t Vector[n]; /* 5个成员的数组类型 */

    n = 7;

    Vector a1;                  /* 5个成员的数组类型 */

    uint16_t a2[n];             /* 7个成员的数组类型 */
}
```

示例

可变长度数组的使用都违背此规则。这些示例显示了由于使用它们可能引起的一些不确定行为。

```
void f(int16_t n)
{
    uint16_t vla[n];    /* 违规 - 如果 n <= 0, 则行为未定义 */
}
void g(void)
{
    f(0);               /* 未定义   */
    f(-1);              /* 未定义   */
    f(10);              /* 行为确定 */
}

void h(uint16_t n,  uint16_t a[10][n])  /* 违规 */
{
    uint16_t (*p)[20];

    /* 未定义 - 除非 n == 20, 否则类型不兼容    */
    p = a;
}
```

参阅

Rule 13.6

8.19 重叠存储(Overlapping storage)

Rule 19.1 不得将对象赋值或复制给重叠的对象

C90 [Undefined 34, 55], C99 [Undefined 51, 94]

级别 强制

分析 不可判定，系统范围

适用 C90，C99

原理

当创建两个对象在内存中有一些重叠并且一个对象被赋值或复制给另一个对象时，该行为是不确定的。

例外

以下情况由于其行为是明确定义的，因此是被允许的：

1. 完全重叠且具有兼容类型的两个对象之间的分配(忽略其类型限定符)
2. 使用标准库函数在存储在部分或完全重叠的对象之间复制

示例

此示例还违反了 Rule 19.2，因为它使用了联合体。

```
void fn(void)
{
    union
    {
        int16_t i;
        int32_t j;
    } a = { 0 }, b = { 1 };

    a.j = a.i;              /* 违规          */
    a = b;                  /* 合规 - 例外 1 */
}

#include <string.h>

int16_t a[20];

void f(void)
{
    memcpy(&a[5], &a[4], 2u * sizeof(a[0]));    /* 违规 */
}

void g(void)
```

```
{
    int16_t *p = &a[0];
    int16_t *q = &a[0];

    *p = *q;            /* 合规 - 例外 1 */
}
```

参 阅

Rule 19.2

Rule 19.2　不得使用 union 关键字

C90 [Undefined 39, 40; Implementation 27], C99 [Unspecified 10; Undefined 61, 62]

级别　建议

分析　可判定，单一编译单元

适用　C90，C99

原理

一个联合体成员可以被修改，然后可以以明确定义的方式回读同一成员。

但是，如果写入了一个工会成员，然后又读回了另一个工会成员，则行为取决于成员的相对大小：

- 如果读取的成员比写入的成员宽，则其值不确定。
- 否则，其值是由实现定义的(取决于编译环境和其配置)。

C 标准允许通过访问类型为无符号字符数组的另一个成员的方式访问联合体成员的字节。但是，由于未指定值的字节也可以被访问，因此不应使用联合。

如果不遵循此规则，则下面几类行为需要被明确确定：

- 填充方式 - 联合体的末尾插入了多少填充；
- 对齐方式 - 联合体内的结构体成员如何对齐；
- 字节序 - 大端模式还是小端模式；
- 位序 - 位在字节内是何顺序，以及如何分配位域。

示例

在此违规示例中，将 16 位值存储到联合中，但回读 32 位值，导致返回未指定的值。

```
uint32_t zext(uint16_t s)
{
    union
    {
        uint32_t ul;
```

```
        uint16_t us;
    } tmp;

    tmp.us = s;
    return tmp.ul;  /* 不确定的值 */
}
```

参 阅

Rule 19.1

8.20 预处理指令(Preprocessing directives)

Rule 20.1　#include 指令之前仅允许出现预处理指令或注释

C90 [Undefined 56], C99 [Undefined 96, 97]

级别　建议

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则应在预处理发生之前应用于文件内容。

原理

为了提高代码的可读性，应将特定代码文件中的所有#include 伪指令组合在一起，放在文件顶部附近。此外，在声明或定义中使用#include 包含标准头文件，或者在包含相关标准头文件之前使用标准库的部分内容，会导致未定义的行为。

示例

```
/* f.h */
xyz = 0;

/* f.c */
int16_t
#include "f.h"      /* 违规 */

/* f1.c */
#define F1_MACRO
#include "f1.h"     /* 合规 */
#include "f2.h"     /* 合规 */
```

```
int32_t i = 0;

#include "f3.h"      /* 违规 */
```

Rule 20.2 头文件名中不得出现“'”、“"”、“\”、字符以及“/*”或“//”字符序列

单引号、双引号、反斜杠，以及 /*或//字符序列

C90 [Undefined 14], C99 [Undefined 31]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

发生下面情况时，行为无法确定：

- 引用头文件预处理指令中的“<”和“>”分隔符之间出现“'”、“\”、“"”字符，或者“/*”或“//”字符序列；
- 引用头文件预处理指令中的“"”分隔符之间出现“'”、“\”字符，或者“/*”或“//”字符序列；

示例

```
#include "fi'le.h" /* 违规 */
```

Rule 20.3 #include 指令后须跟随<filename>或"filename"序列

C90 [Undefined 48], C99 [Undefined 85]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

此规则在执行宏替换后适用。

原理

#include 指令必须是下面形式之一，否则其行为不确定：

- #include <filename>
- #include "filename"

示例

```
#include "filename.h"       /* 合规 */
#include <filename.h>       /* 合规 */
#include another.h          /* 违规 */

#define HEADER "filename.h"
#include HEADER             /* 合规 */
#define FILENAME file2.h
#include FILENAME           /* 违规 */

#define BASE "base"
#define EXT ".ext"
#include BASE EXT           /* 违规 - 字符串在预处理后被串联    */

#include "./include/cpu.h"  /* 合规 - 文件名可以包含路径        */
```

Rule 20.4　宏不得与关键字同名

C90 [Undefined 56], C99 [Undefined 98]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

此规则适用于所有关键字，包括那些实现语言扩展的关键字。

原理

使用宏更改关键字的含义可能会造成混淆。在定义宏的名称与关键字相同的情况下包含标准库头文件，其行为未定义。

示例

下面的违规示例中，更改了 int 关键字的行为。在存在此宏的情况下包含标准标头会导致未定义的行为。

```
#define int some_other_type
#include <stdlib.h>
```

下面的示例中，重新定义关键字 while 违规，但定义扩展为语句的宏是合规的。

```
#define while(E) for (; (E);)   /* 违规 - while 被重定义    */
#define unless(E) if (!(E))     /* 合规                     */

#define seq(S1, S2) do { \
    S1; S2; } while (false)     /* 合规                     */
#define compound(S) { S; }      /* 合规                     */
```

下面示例在 C90 中合规，但在 C99 中违规，因为inline 不是 C90 中的关键字。

```
/* 如果针对C90进行编译，则删除inline关键字 */
#define inline
```

参 阅

Rule 21.1

Rule 20.5 不应使用#undef

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

原理

#undef 的使用可能导致我们不清楚翻译单元中的特定点上存在哪些宏。

示例

```
#define QUALIFIER volatile
#undef QUALIFIER                /* 违规 */
void f(QUALIFIER int32_t p)
{
    while (p != 0)
    {
        ; /* 等待... */
    }
}
```

Rule 20.6 看起来像预处理指令的符号不得出现在宏参数内

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

包含令牌序列的参数(可能在其他情况下将充当预处理指令)将导致未定义的行为。

示例

```
#define M(A) printf (#A)

#include <stdio.h>

void main(void)
```

```
{
    M (
#ifdef SW               /* 违规 */
        "Message 1"
#else                   /* 违规 */
        "Message 2"
#endif                  /* 违规 */
    );
}
```

上面的代码可能会打印出：

```
#ifdef SW "Message 1" #else "Message 2" #endif
```

或者

```
"Message 2"
```

或者表现出其他一些行为。

Rule 20.7　宏参数展开产生的表达式应放在括号内

[Koenig 78－81]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

如果任何宏参数的扩展产生形成一个表达式的令牌或令牌序列，则该表达式在完全扩展的宏中应：

- 本身成为带括号的表达式；或者
- 用括号括起来。

小贴士：这不要求所有宏参数都用括号括起来；可以在宏参数中提供括号。

原理

如果不使用括号，则在发生宏替换时，因为运算符优先级的原因，可能无法获得预期的结果。

如果没有将表达式作为宏参数，则不需要括号，因为不涉及运算符。

示例

违规示例：

```
#define M1(x, y)  (x * y)

r = M1 (1 + 2, 3 + 4);
```

宏展开后，得到

```
r = ( 1 + 2 * 3 + 4 );
```

表达式 1 + 2 和3 + 4 分别来自参数 x 和 y 的扩展，但都不包含在括号中。表达式的最终结果为 11，而非预期的 21。

我们可以通过将宏定义时将参数用括号括起来，或者在宏调用时加入括号的方法，来达成合规目的，例如

```
r = M1((1 + 2), (3 + 4));            /* 合规 */

#define M2(x, y) (( x ) * ( y ))

r = M2 (1 + 2, 3 + 4);               /* 合规 */
```

下面的合规示例中，x 的第一个展开是##运算符的操作数，该运算符不产生表达式；x 的第二个展开是一个表达式，根据需要将其括起来。

```
#define M3(x) a ## x = (x)

int16_t M3(0);
```

下面的合规示例中，参数M 作为成员名称的扩展不会产生表达式。 扩展参数 S 会产生一个带有结构或联合类型的表达式，该表达式确实需要括号。

```
#define GET_MEMBER(S, M) (S).M

v = GET_MEMBER (s1, minval);
```

下面的合规示例表明，用括号括起宏参数不总是必需的，尽管它通常是遵守此规则的最简单方法。

```
#define F(X) G(X)
#define G(Y) ((Y) + 1)

int16_t x = F(2);
```

示例中的宏完全展开后得到((2)+1)。追溯宏展开，值 2 由宏 G 中参数 Y 的展开产生，而宏 G 中的参数X 又由宏F 中的参数X 产生。由于 2 被括在完全展开的宏中，因此代码是合规的。

参 阅

Dir 4.9

Rule 20.8 #if 或#elif 预处理指令的控制表达式的计算结果应为 0 或1

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

此规则不适用于预处理指令中未求值的表达式。如果控制表达式位于要排除的代码内，并且不会影响是否排除代码，则不对控制表达式求值。

原理

强类型要求条件预处理指令的控制表达式应具有布尔值。

示例

```
#define FALSE 0
#define TRUE  1

#if FALSE       /* 合规                      */
#endif

#if 10          /* 违规                      */
#endif

#if !defined(X) /* 合规                      */
#endif

#if A > B       /* 合规 - 假定A和B都是数字   */
#endif
```

参 阅

Rule 14.4

Rule 20.9　#if 或#elif 预处理指令的控制表达式中使用的所有标识符应在其评估前被#define 定义

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

除了使用#define 预处理指令外，标识符还可以通过其他实现定义的方式有效地#define。例如，某些实现支持：

- 使用编译器命令行选项，例如“-D”，以允许在翻译之前定义标识符；
- 使用环境变量来达到相同的效果；
- 编译器提供的预定义标识符。

原理

如果试图在预处理器指令中使用宏标识符，但尚未定义该标识符，则预处理器将假定其值为零。 这可

能不符合开发人员的期望。

示例

下面的示例中，假定宏M 未定义。

```
#if M == 0       /* 违规                                  */
                 /* 此处不确定"M"展开为零还是未定义       */
#endif

#if defined(M)   /* 合规 - M 并未被评估                   */
  #if M == 0     /* 合规 - 已知 M 被定义                  */
                 /* 此处 M 展开后一定为 0                 */
  #endif
#endif

/* 合规 - 仅在确定B被定义后才会在 (B == 0) 被求值         */
#if defined(B) && (B == 0)
#endif
```

Rule 20.10 不应使用“#”和“##”预处理运算符

C90 [Unspecified 12; Undefined 51, 52], C99 [Unspecified 25; Undefined 3, 88, 89]

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

原理

多个“#”运算符，多个“##”运算符或者“#”运算符与“##”运算符混合出现的预处理程序中，其运算符评估顺序在 C 语言中未明确指定。因此，在某些情况下，我们无法预测宏展开的结果。

使用“##”运算符可能会导致代码晦涩难懂。

小贴士：规则 1.3 涵盖了以下情况之一时发生的未定义行为：

- *“#”运算符的结果不是有效的字符串；*
- *“##”运算符的结果不是有效的预处理令牌。*

参阅

Rule 20.11

Rule 20.11 紧跟在“#”运算符之后的宏参数后面不得紧随“##”运算符

C90 [Unspecified 12], C99 [Unspecified 25]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

C 语言未指定多个“#”运算符，多个“##”运算符或者“#”与“##”混合出现的预处理程序的评估顺序。Rule 20.10 不建议使用“#”和“##”。特别是，“#”运算符的结果是字符串，将其粘贴到任何其他预处理令牌中都极可能产生无效的令牌。

示例

```
#define A(x)         #x       /* 合规 */
#define B(x, y)      x ## y   /* 合规 */
#define C(x, y)      #x ## y /* 违规 */
```

参阅

Rule 20.10

Rule 20.12 用作“#”或“##”运算符的操作数的宏参数，不得是本身需要进一步宏替换的操作数

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

用作“#”或“##”运算符的操作数的宏参数在使用前不会展开。替换文本中其他位置出现的相同参数则会被展开。如果宏参数本身需要进行宏替换，则在宏替换内的混合上下文中使用它可能无法满足开发人员的期望。

示例

在下面的违规示例中，宏参数 x 会被替换为 AA，当不用作“##”的操作数时，将进行进一步的宏替换。

```
#define AA 0xffff
#define BB(x) (x) + wow ## x    /* 违规      */

void f(void)
{
    int32_t wowAA = 0;
```

```
    /* 展开后为 wowAA = (0xffff) + wowAA;    */
    wowAA = BB(AA);
}
```

在下面的合规示例中，宏参数 X 不需要进一步的宏替换。

```
int32_t speed;
int32_t speed_scale;
int32_t scaled_speed;

#define SCALE(X) ((X) * X ## _scale)

/* 展开为 scaled_speed = ((speed) * speed_scale); */
scaled_speed = SCALE(speed);
```

Rule 20.13 以“#”作为第一个字符的一行代码应为有效的预处理指令

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

“#”和预处理指令令牌之间允许有空格。

原理

我们可以使用预处理器指令有条件地排除源代码，直到遇到相应的#else，#elif 或#endif 指令为止。编译器可能不能检测到包含在排除的源代码中的格式错误的或无效的预处理指令，这可能会导致排除了比预期更多的代码。

采用“要求所有预处理器指令在语法上均有效，即使它们出现在排除的代码块中”的方法，可确保不发生这种情况。

示例

在下面示例中，如果定义了 AAA，则#ifndef 和#endif 直接指令之间的所有代码都可以排除；开发人员打算将 AAA 分配给 x，但是#else 指令输入错误，并且未被编译器诊断。

```
#define AAA 2

int32_t foo(void)
{
    int32_t x = 0;
```

```
#ifndef AAA
    x = 1;
#else1          /* 违规 */
    x = AAA;
#endif

    return x;
}
```

下面的合规示例中，注释中出现的文本#start 不是预处理指令令牌。

```
/*
#start is not a token in a comment
*/
```

Rule 20.14 所有#else，#elif 和#endif 预处理程序指令都应和与其相关的#if，#ifdef 或#ifndef 指令位于同一文件中

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

原理

当使用条件编译指令来包含或排除代码块，而这些条件编译指令分布在多个文件中时，可能会造成混乱。在同一文件中终止#if 指令，可以减少代码的视觉复杂度以及维护期间出错的可能性。

小贴士：#if 指令可以在被包含的文件中使用，前提是它们在同一文件中终止。

示例

```
/* file1.c */
#ifdef A                /* 合规 */
#include "file1.h"
#endif
/* End of file1.c */

/* file2.c */
#if 1                   /* 违规 */
#include "file2.h"
/* End of file2.c*/

/* file1.h */
#if 1                   /* 合规 */
#endif
```

```
/* End of file1.h */

/* file2.h */
#endif
/* End of file1.h */
```

8.21 标准库(Standard libraries)

Rule 21.1 不得将#define 和#undef 用于保留的标识符或保留的宏名称

C90 [Undefined 54, 57, 58, 62, 71]

C99 [Undefined 93, 100, 101, 104, 108, 116, 118, 130]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

此规则适用于以下情况：

- 标识符或宏名称以下划线开头；
- C 标准第 7 章“库(Libraries)”中描述的文件范围中的标识符；
- C 标准第 7 章“库(Libraries)”中描述的标准头文件中的宏名称。

此规则还禁止在已定义的标识符上使用#define 或#undef，因为这会导致未明确定义的行为。

此规则不适用于的C 标准中标题为“ Future Library Directions”的章节中描述的标识符或宏名称。

C 标准中，未包含标准头文件时，定义与下列标识符相同的宏是被允许的：

- 标准头文件中定义的宏；
- 标准头文件中声明的具有文件范围的标识符。

但是，此规则中不允许这种做法，因为这可能会引起混淆。

小贴士：宏 NDEBUG 没有在标准标头文件中定义，因此可以被#define。

原理

保留的标识符和保留的宏名称旨在供实现使用。 删除或更改保留宏的含义可能会导致不确定的行为。

示例

```
#undef __LINE__              /* 违规 - "_"开头                    */
#define _ GUARD_H 1          /* 违规 - "_"开头                    */
#undef _BUILTIN_sqrt         /* 违规 - 该实现可能将_BUILTIN_sqrt
```

```
                          * 用于其他目的，例如生成sqrt指令    */

#define defined             /* 违规 - 保留标识符                */
#define errno my_errno      /* 违规 - 库标识符                  */
#define isneg(x) ((x) < 0)  /* 合规 - 此规则不适用于
                             *《future library directions》      */
```

参 阅

Rule 20.4

Rule 21.2　不得声明保留的标识符或宏名称

C90 [Undefined 5 7, 58, 64, 71], C99 [Undefin ed 93, 100, 101, 104, 108, 116, 118, 130]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

有关相关标识符和宏名称的说明，请参见 Rule 21.1 的“展开”。

原理

C 标准允许实现依赖于保留标识符的行为，并且可以对其进行特殊对待。如果保留的标识符被重用，则程序可能会表现出不确定的行为。

示例

在下面违规示例中，函数 memcpy 被显式声明。声明此功能的合规方法是包含<string.h>。

```
/*
 * 包含 <stddef.h> 以定义 size_t
 */
#include <stddef.h>

extern void *memcpy (void *restrict s1, const void *restrict s2, size_t n);
```

除了库函数本身之外，实现为每个标准库函数提供类似函数的宏定义是被允许的。编译器编写者经常使用此功能来生成有效的内联操作，以代替对库函数的调用。使用类似函数的宏，可以将对库函数的调用替换为对保留函数的调用，该调用在编译器的代码生成阶段检测到并由内联操作替换。例如，可以使用函数式宏来编写声明<math.h>中的段 sqrt，该宏会生成对_BUILTIN_sqrt 的调用，该调用将在支持它的处理器上被内联 SQRT 指令替换：

```
extern double sqrt(double x);
```

```
#define sqrt(x) (_BUILTIN_sqrt(x))
```

下面违规示例中的代码可能会干扰编译器用于处理 sqrt 的内置机制，因而产生不确定的行为：

```
#define _BUILTIN_sqrt(x) (x)        /* 违规                                  */
#include <math.h>

float64_t x = sqrt((float64_t)2.0); /* sqrt可能具备C标准中定义的行为     */
```

Rule 21.3　不得使用<stdlib.h>中的内存分配和释放函数

C90 [Unspecified 19; Undefined 9, 91, 92; Implementation 69]

C99 [Unspecified 39, 40; Undefined 8, 9, 168 – 171; Implementation J.3.12(35)]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

不得使用标识符 calloc，malloc，realloc 和 free，并且不得展开含有这些名称的宏。

原理

使用标准库提供的动态内存分配和释放例程很容易导致不确定的行为，例如：

- 释放未被动态分配的内存；
- 以任何方式使用指向已释放的内存的指针；
- 在将值存储到分配的内存之前访问该内存。

小贴士：此规则是 Dir 4.12 的特定实例。

参阅

Dir 4.12, Rule 18.7, Rule 22.1, Rule 22.2

Rule 21.4　不得使用标准头文件<setjmp.h>

C90 [Unspecified 14; Undefined 64 – 67]

C99 [Unspecified 32; Undefined 118 – 121, 173]

[Koenig 74]

级别　必要

分析　可判定，单一编译单元

适用　C90，C99

展开

<setjmp.h>中指定的任何功能均不得使用。

原理

setjmp 和 longjmp 允许绕过正常的函数调用机制。使用它们可能导致不确定的行为。

Rule 21.5 不得使用标准头文件<signal.h>

C90 [Undefined 67 - 69; Implementation 48 - 52]

C99 [Undefined 122 - 127; Implementation J.3.12(12)]

[Koenig 74]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

<signal.h>中指定的任何功能均不得使用。

原理

信号处理包含实现定义和未定义的行为。

Rule 21.6 不得使用标准库输入/输出函数

C90 [Unspecified 2 - 5, 16 - 18; Undefined 77 - 89; Implementation 53 - 68]

C99 [Unspecified 3 - 6, 34 - 37; Undefined 138 - 166, 186; Implementation J.3.12(14 - 32)]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

此规则适用于<stdio.h>指定的功能，以及在 C99 标准的第 4.24.2 和 7.24.3 节描述的<wchar.h>指定的宽字符等效项。

这些文件里的标识符均不得使用，并且不得展开含有这些标识符的宏。

原理

文件和文件流的 I/O 操作具有未指定，未定义和实现定义的行为。

参阅

Rule 22.1, Rule 22.3, Rule 22.4, Rule 22.5, Rule 22. 6

Rule 21.7 不得使用<stdlib.h>中的 atof、atoi、atol 和 atoll 函数

C90 [Undefined 90], C99 [Undefined 113]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

不得使用 atof、atoi、atol 和 atoll(仅 C99)标识符，并且不得展开含有这些标识符的宏。

原理

当无法转换字符串时，这些函数具有未定义的行为。

Rule 21.8 不得使用<stdlib.h>中的 abort, exit, getenv 和 system 函数

C90 [Undefined 93; Im plementation 70－73]

C99 [Undefined 172, 174, 175; Implementation J.3.12(36－38)]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

不得使用 abort, exit, getenv 和 system 标识符，并且不得展开含有这些标识符的宏。

原理

这些函数具有未定义和与实现相关的行为。

Rule 21.9 不得使用<stdlib.h>中的 bsearch 和 qsort 函数

C90 [Unspecified 20, 21], C99 [Unspecified 41, 42; Undefined 176－178]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

不得使用 bsearch 和 qsort 标识符，并且不得展开含有这些标识符的宏。

原理

如果比较函数在比较元素时行为不一致，或者修改某些元素，其行为是未定义的。

小贴士：可以通过确保比较函数从不返回 0 来避免与比较相等的元素有关的未指定行为。当两个元素相等时，比较函数可以返回在在初始数组中表示其相对顺序的值。

qsort 的实现很可能是递归的，因此会对堆栈资源提出未知要求。在嵌入式系统中，这是个问题，因为堆栈可能是固定的，通常很小。

Rule 21.10 不得使用标准库时间和日期功能

C90 [Unspecified 20, 21], C99 [Unspecified 41, 42; Undefined 176－178]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

<time.h>指定的任何功能都不得使用。

在 C99 中，不得使用标识符 wcsftime，并且不得展开含有该名称的宏。

原理

时间和日期函数具有未指定，未定义和实现定义的行为。

Rule 21.11 不得使用标准头文件<tgmath.h>

C99 [Undefined 184, 185]

级别 必要

分析 可判定，单一编译单元

适用 C90，C99

展开

<tgmath.h>指定的任何功能都不得使用。

原理

使用<tgmath.h>的功能可能会导致不确定的行为。

示例

```
#include <tgmath.h>
```

```
float f1, f2;

void f(void)
{
    f1 = sqrt(f2);      /* 违规 - 通用版 sqrt 函数禁止使用  */
}

#include <math.h>

float f1, f2;

void f(void)
{
    f1 = sqrtf(f2);     /* 合规 - 可使用 float 版 sqrt 函数 */
}
```

Rule 21.12 不得使用<fenv.h>的异常处理功能

C99 [Unspecified 27, 28; Undefined 109－111; Implementation J.3.6(8)]

级别 建议

分析 可判定，单一编译单元

适用 C90，C99

展开

不得使用标识符feclearexcept,fegetexceptflag,feraiseexcept,fesetexceptflag和fetestexcept，并且不得展开含有这些名称的宏。

不得使用宏 FE_INEXACT，FE_DIVBYZERO，FE_UNDERFLOW，FE_OVERFLOW，FE_INVALID 和 FE_ALL_EXCEPT，以及任何实现定义的浮点异常宏。

原理

在某些情况下，浮点状态标志的值是未指定的，尝试访问它们可能导致不确定的行为。

feraiseexcept 函数引发异常的顺序未指定，因此可能导致为特定顺序设计的程序无法正确运行。**示例**

```
#include <fenv.h>

void f(float32_t x, float32_t y)
{
    float32_t z;

    feclearexcept(FE_DIVBYZERO);        /* 违规 */
```

```
    z = x / y;

    if (fetestexcept(FE_DIVBYZERO))     /* 违规 */
    {
    }
    else
    {
#pragma STDC FENV_ACCESS ON

        z = x * y;
    }

    if (z > x)
    {
#pragma STDC FENV_ACCESS OFF

        if (fetestexcept(FE_OVERFLOW))  /* 违规 */
        {
        }
    }
}
```

8.22 资源(Resources)

本节中的许多规则仅在背离其他节中的规则时适用。

Rule 22.1 通过标准库功能动态获取的所有资源均应明确释放

级别 必要

分析 不可判定，系统范围

适用 C90，C99

展开

分配资源的标准库函数包括 malloc，calloc，realloc 和 fopen。

原理

如果未明确释放资源，则可能由于这些资源的耗尽而发生故障。尽快释放资源可以减少耗尽的可能性。

示例

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

```
int main(void)
{
    void *b = malloc(40);

    /* 违规 - 动态分配的内存未被释放                */
    return 1;
}

#include <stdio.h>

int main(void)
{
    FILE *fp = fopen("tmp", "r");

    /* 违规 - 文件未被关闭                          */
    return 1;
}
```

在下面的不兼容示例中，打开“tmp-2”时，“tmp-1”上的句柄丢失。

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

    fp = fopen("tmp-1", "w");

    fprintf(fp, "*");

    /* 文件 "tmp-1" 应在此处关闭，但数据流漏掉了它  */
    fp = fopen("tmp-2", "w");

    fprintf(fp, "!");

    fclose(fp);

    return(0);
}
```

参阅

Dir 4.12, Dir 4.13, Rule 21.3, Rule 21.6

Rule 22.2　只有通过标准库函数分配的内存块才能释放

C90 [Undefined 92], C99 [Undefined 169]

级别　强制

分析　不可判定，系统范围

适用　C90，C99

展开

分配内存的标准库函数是 malloc，calloc 和 realloc。

当一个内存块的地址传递给 free 时，它会被释放；当它的地址传递给 realloc 时，它可能会被隐式的释放。一旦释放，就不再认为已分配了内存块，因此无法随后再次释放。

原理

释放未分配的内存，或多次释放相同的已分配内存会导致未定义的行为。

示例

```
#include <stdlib.h>

void fn(void)
{

    int32_t a;

    /* 违规 - a 并未指向分配的内存      */
    free(&a);
}

void g(void)
{
    char *p =(char *)malloc(512);
    char *q = p;

    free(p);

    /* 违规 - 分配的块二次释放          */
    free(q);

    /* 违规 - 分配的块可能会被三次释放  */
    p =(char *)realloc(p, 1024);
}
```

参阅

Dir 4.12, Dir 4.13, Rule 21.3

Rule 22.3　不得在不同的数据流上同时打开同一文件以进行读写访问

C90 [Implementation 61], C99 [Implementation J.3.12(22)]

级别　必要

分析　不可判定，系统范围

适用　C90，C99

展开

此规则适用于使用标准库功能打开的文件。它也可以应用于执行环境提供的类似功能。

原理

C 标准未指定通过不同的流写入和读取文件的行为。

小贴士：以只读访问时，可以多次打开文件。

示例

```
#include <stdio.h>

void fn(void)
{
    FILE *fw = fopen("tmp", "r+");  /* "r+" 以读/写打开 */
    FILE *fr = fopen("tmp", "r");   /* 违规             */
}
```

参 阅

Rule 21.6

Rule 22.4　禁止尝试对以只读方式打开的流执行写操作

级别　强制

分析　不可判定，系统范围

适用　C90，C99

原理

C 标准未指定尝试向只读流执行写操作的行为。因此，向只读流执行写操作被认为是不安全的。

示例

```
#include <stdio.h>

void fn(void)
{
    FILE *fp = fopen("tmp", "r");
```

```
    (void)fprintf(fp, "What happens now?");     /* 违规 */

    (void)fclose(fp);
}
```

参 阅

Rule 21.6

Rule 22.5 禁止反引用指向 FILE 对象的指针

级别 强制

分析 不可判定，系统范围

适用 C90，C99

展开

指向 FILE 对象的指针不得直接或间接反引用(例如，通过调用 memcpy 或 memcmp)。

原理

C 标准(C90 第 7.9.3(6)节，C99 第 7.9.3(6)节)指出，用于控制流的 FILE 对象的地址可能很重要，并且该对象的副本可能不会具有相同的行为。此规则旨在确保不制作此类副本。

禁止直接操作FILE 对象，因为这可能与将其用作流指示符不兼容。

示例

```
#include <stdio.h>

FILE *pf1;
FILE *pf2;
FILE f3;

pf2 = pf1;      /* 合规 */
f3 = *pf2;      /* 违规 */
```

下面的示例假定 FILE *指定了一个名为 pos 的成员的完整类型：

```
pf1->pos = 0;   /* 违规 */
```

参 阅

Rule 21.6

Rule 22.6 关联的流关闭后，禁止再使用指向 FILE 的指针值

级别 强制

分析 不可判定，系统范围

适用 C90，C99

原理

C 标准指出，在对流执行关闭操作后，FILE 指针的值不确定。

示例

```
void fn(void)
{
    FILE *fp;
    void *p;

    fp = fopen("tm p", "w");

    if(fp == NULL)
    {
        error_action();
    }

    fclose(fp);

    fprintf(fp, "?");   /* 违规 */
    p = fp;             /* 违规 */
}
```

参阅

Dir 4.13, Rule 21.6

7 参考文献(References)

[1] MISRA Guidelines for the Use of the C Language In Vehicle Based Software, ISBN 0-9524159-9-0, Motor Industry Research Association, Nuneaton, April 1998

[2] ISO/IEC 9899: 1990, Programming languages — C, International Organizati on for Standardization, 1990

[3] Hatton L., Safer C - Developing Software for High-integrity and Safety-critical Systems, ISBN 0-07-707640-0, McGraw-Hill, 19 94

[4] ISO/IEC 9899:1990/COR 1:1995, Technical Corrigendum 1, 1995

[5] ISO/IEC 9899:1990/AMD 1:1995, Amendment 1, 1995

[6] ISO/IEC 9899: 1990/COR 2:1996, Technical Corrigendum 2, 1996

[7] ANSI X3.159-1989, Programming languages — C, American National Standards Institute, 1989

[8] ISO/IEC 9899:1999, Programming languages — C, International Organiz ation for Standardization, 1999

[9] ISO/IEC 9899:1999/COR 1:2001 , Technical Corrigendum 1, 2001

[10]ISO/IEC 9899:199 9/COR 2:2004, Technical Corrigendum 2, 2004

[11]ISO/IEC 9899:199 9/COR 3:2007, Technical Corrigendum 3, 2007

[12] ISO/IEC 9899:1999 Committee Draft WG14/N1256, Programming languages — C, International Organization for Standardization, September 2007

[13] ISO/IEC 9899:2011, Programming languages — C, Internationa l Organization for Standardization, 2011

[14] ISO/IEC 9899:2011/C OR 1:2012, Technical Corrigendum 1, 2012

[15] MISRA Development G uidelines for Vehicle Based Software, ISBN 0-9524156-0-7, Mo tor Industry Research Association, Nuneaton, November 1994

[16] MISRA AC AGC Guidelines for the application of MISRA-C:2004 in the context of automatic code generatio n, ISBN 978-1-906400-02-6, MIRA Limited, Nuneaton, November 2007

[17] MISRA AC GMG Generic modelling design and style guidelines, ISBN 978-1-906400-06-4, MIRA Limited, Nuneaton, May 2009

[18] MI SRA AC SLSF Modelling design and style guidelines for the application of Simulink and Stateflow, ISBN 978-1-906400-07-1, MIRA Limited, Nun eaton, May 2009

[19] MISRA AC TL Modelling style guidelines for the application of Targetlink in the context of automatic code generation, ISBN 978-1-906400-01-9, MIRA Limi ted, Nuneaton, November 2007

[20]CRR80, The Use of Commercial Off -the-Shelf (COTS) Software in Safety Related Application s, ISBN 0-7176-0984-7, HSE Books

[21] ISO 9001:2008, Quality management systems — Requirements, International Organization for Standardization, 2008

[22] ISO 90003:20 04, Software engineering — Guidelines for the application of ISO 9001:2000 to computer software, ISO, 2004

[23] ISO 26262:2011, Road vehicles — Functional safety, ISO, 2011

[24] DO-178C/ED-1 2C, Software Considerations in Airborne Systems and Equipment Certification,

RTCA, 2011

[25] The TickIT Guide, Using ISO 9001: 2000 for Software Quality Management System Construction, Certification and Continual Improvement, Issue 5, British Standards Institution, 2001

[26]Straker D., C Style: Standards and Guidelines, ISBN 0－13-116898-3, Prentice Hall 1991

[27]Fenton N.E. and Pfleeger S.L., Software Metrics: A Rigorous and Practical Approach, 2nd Edition, ISBN 0-534-95429-1, PWS, 1998

[28] MISR A Report 5 Software Metrics, Motor Industry Research Association, Nuneaton, February 1995

[29] MISR A Report 6 Verification and Validation, Motor Industry Research Association, Nuneaton, February 1995

[30] Kernighan B.W., Ritchie D.M., The C programming language, 2nd edition, ISBN 0-13-110362-8, Prentice Hall, 1988 (note: The 1st edition is not a suitable reference document as it does not describe ANSI/ISO C)

[31] Koenig A., C Traps and Pitfalls, ISBN 0-201-17928-8, Add ison-Wesley, 1988

[32] IEC 61508:2010, Functional safety of electrical/electronic/programmable electronic safety-related systems, Int ernational Electromechanical Commission, in 7 parts published in 2010

[33] EN 50128:2011, Railway a pplications — Communications, signalling and processing systems — Software for railway control and protec tion, CENELEC, 2011

[34] IEC 62304:2006, Medical device software — Software life cycle processes, IEC, 2006

[35]ANSI/IEEE Std 754, IEEE Standard for Binary Floating-Point Arithmetic, 1985

[36] ISO/IEC 10646:2003, Informat ion technology — Universal Multiple-Octet Coded Character Set (UCS), Inter national Organization for Standardization, 2003

[37] Goldberg D., What Every Computer Scientist Should Know about Floating-Point Arithmetic, Computing Surveys, March 1991

[38] Software Engin eering Center, Information-technology Promotion Agency, Japan (IPA/SEC), Embedded System development Coding Reference (ESCR) [C language edition] Version 1.1, SEC Books, 2012

附录 A：准则摘要(Summary of guidelines)

实施/实现(The implementation)

Dir 1.1	必要	程序输出所依赖的任何由实现定义的行为都应被记录和理解

编译与构建(Compilation and build)

Dir 2.1	必要	所有源文件都应通过编译且没有任何编译错误

需求可追溯性(Requirements traceability)

Dir 3.1	必要	所有代码应可追溯到书面要求

代码设计(Code Design)

Dir 4.1	必要	所有代码应可追溯到书面要求
Dir 4.2	建议	汇编语言的所有运用都应记录在案
Dir 4.3	必要	汇编语言应被封装和隔离
Dir 4.4	建议	代码段不应被“注释掉”
Dir 4.5	建议	具有相同可见性的相同名称空间中的标识符在印刷/屏幕显示上应明确
Dir 4.6	建议	应使用指示大小和符号的 typedef 类型代替基本数字类型
Dir 4.7	必要	如果函数返回错误信息，则应测试该错误信息
Dir 4.8	建议	如果一个指向结构体或联合体的指针在编译/解释时从未被反引用，则应隐藏该对象的实现
Dir 4.9	建议	应该优先使用函数，而不是类似函数的宏(如果它们可以互换)
Dir 4.10	必要	应采取预防措施以防止头文件的内容被多次包含
Dir 4.11	必要	应检查传递给库函数的值的有效性
Dir 4.12	必要	不得使用动态内存分配
Dir 4.13	建议	用于对资源进行操作的功能应按适当的顺序调用

标准 C 环境(A standard C environment)

Rule 1.1	必要	程序不得违反标准 C 语法和约束，并且不得超出具体实现的编译限制
Rule 1.2	建议	不应该使用语言扩展
Rule 1.3	必要	不得发生未定义或严重的未指定行为

未使用的代码(Unused code)

Rule 2.1	必要	项目不得包含不可达代码(unreachable code)
Rule 2.2	必要	不得有无效代码(dead code)
Rule 2.3	建议	项目不应包含未被使用的类型(type)声明
Rule 2.4	建议	项目不应包含未被使用的类型标签(tag)声明
Rule 2.5	建议	项目不应包含未被使用的宏(macro)声明
Rule 2.6	建议	函数不应包含未被使用的执行标签(label)声明
Rule 2.7	建议	函数中不应有未使用的变量

注释(Comments)

Rule 3.1	必要	字符序列“/*”和“//”不得在注释中使用
Rule 3.2	必要	“//”注释中不得使用换行(即“//”注释中不得使用行拼接符“\”)

字符集和词汇约定(Character sets and lexical conventions)

Rule 4.1	必要	八进制和十六进制转译序列应有明确的终止识别标识
Rule 4.2	建议	禁止使用三字母词(trigraphs)

标识符(Identifiers)

Rule 5.1	必要	外部标识符不得重名
Rule 5.2	必要	同范围和命名空间内的标识符不得重名
Rule 5.3	必要	内部声明的标识符不得隐藏外部声明的标识符
Rule 5.4	必要	宏标识符不得重名
Rule 5.5	必要	宏标识符与其他标识符不得重名
Rule 5.6	必要	typedef 名称应是唯一标识符
Rule 5.7	必要	标签(tag)名称应是唯一标识符
Rule 5.8	必要	全局(external linkage)对象和函数的标识符应是唯一的
Rule 5.9	建议	局部全局(internal linkage)对象和函数的标识符应是唯一的

类型(Types)

Rule 6.1	必要	位域(位带)仅允许使用适当的类型来声明(位域成员类型限制)
Rule 6.2	必要	单比特(single-bit)位域成员不可声明为有符号类型

字符和常量(Literals and constants)

Rule 7.1	必要	禁止使用八进制常数
Rule 7.2	必要	后缀“u”或“U”应使用于所有无符号的整数常量
Rule 7.3	必要	小写字符“l”不得作为常量的后缀使用(仅可使用“L”)
Rule 7.4	必要	除非对象的类型为“指向 const char 的指针”，否则不得将字符串常量赋值给该对象

声明和定义(Declarations and definitions)

Rule 8.1	必要	类型须明确声明
Rule 8.2	必要	函数类型应为带有命名形参的原型形式
Rule 8.3	必要	对象或函数的所有声明均应使用相同的名称和类型限定符
Rule 8.4	必要	全局(external linkage)的对象和函数，应有显式的合规的声明
Rule 8.5	必要	全局对象或函数应在且只在一个文件中声明一次
Rule 8.6	必要	全局标识符应在且只在一处定义
Rule 8.7	建议	仅在本编译单元中调用的对象和函数，应定义成局部属性
Rule 8.8	必要	“static”修饰符应用在所有局部全局对象和局部函数(internal linkage)的声明中
Rule 8.9	建议	若一个对象的标识符仅在一个函数中出现，则应将它定义在块范围内
Rule 8.10	必要	内联函数应使用静态存储类声明
Rule 8.11	建议	声明具有外部链接的数组时，应明确指定其大小
Rule 8.12	必要	在枚举列表中，隐式指定的枚举常量的值应唯一
Rule 8.13	建议	指针应尽可能指向 const 限定类型
Rule 8.14	必要	不得使用类型限定符“restrict”

初始化(Initialization)

Rule 9.1	强制	具有自动存储持续时间的对象(临时变量)的值在设置前不得读取
Rule 9.2	必要	集合或联合体的初始化应括在花括号“{}”中
Rule 9.3	必要	数组不得部分初始化
Rule 9.4	必要	数组的元素不得被初始化超过一次
Rule 9.5	必要	在使用指定初始化方式初始化数组对象的情况下，应明确指定数组的大小

基本类型模型(The essential type model)

Rule 10.1	必要	操作数不得为不适当的基本类型
Rule 10.2	必要	字符类型的表达式不得在加减运算中使用不当
Rule 10.3	必要	表达式的值不得赋值给具有较窄基本类型或不同基本类型的对象
Rule 10.4	必要	执行常规算术转换的运算符的两个操作数应有相同的基本类型
Rule 10.5	建议	表达式的值不应(强制)转换为不适当的基本类型
Rule 10.6	必要	复合表达式的值不得赋值给具有较宽基本类型的对象
Rule 10.7	必要	如果将复合表达式用作执行常规算术转换的运算符的一个操作数，则另一个操作数不得具有更宽的基本类型
Rule 10.8	必要	复合表达式的值不得转换为其他基本类型或更宽的基本类型

指针类型转换(Pointer type conversions)

Rule 11.1	必要	不得在指向函数的指针和任何其他类型的指针之间进行转换
Rule 11.2	必要	不得在指向不完整类型的指针和其他任何类型间进行转换
Rule 11.3	必要	不得在指向不同对象类型的指针之间执行强制转换
Rule 11.4	建议	不得在指向对象的指针和整数类型之间进行转换
Rule 11.5	建议	不得将指向 void 的指针转换为指向对象的指针
Rule 11.6	必要	不得在指向 void 的指针和算术类型之间执行强制转换
Rule 11.7	必要	不得在指向对象的指针和非整数算术类型之间执行强制转换
Rule 11.8	必要	强制转换不得从指针指向的类型中删除任何 const 或volatile 限定符
Rule 11.9	必要	宏“NULL”是整数型空指针常量的唯一允许形式

表达式(Expressions)

Rule 12.1	建议	表达式中运算符的优先级应明确
Rule 12.2	必要	移位运算符的右操作数应在零到比左操作数基本类型的位宽度小一的范围内
Rule 12.3	建议	不得使用逗号(,)运算符
Rule 12.4	建议	常量表达式的求值不应导致无符号整数的回绕

副作用(Side effects)

Rule 13.1	必要	初始化程序列表不得包含持久性副作用
Rule 13.2	必要	在所有合法的评估命令下，表达式的值应与其持续的副作用相同

Rule 13.3	建议	包含自增(++)或自减(--)运算符的完整表达式，除由自增或自减运算符引起的副作用外，不应有其他潜在的副作用
Rule 13.4	建议	不得使用赋值运算符的结果
Rule 13.5	必要	逻辑与(&&)和逻辑或(||)的右操作数不得含有持久性副作用
Rule 13.6	强制	sizeof 运算符的操作数不得包含任何可能产生副作用的表达式

控制语句表达式(Control statement expressions)

Rule 14.1	必要	循环计数器的基本类型不能为浮点型
Rule 14.2	必要	for 循环应为良好格式
Rule 14.3	必要	控制表达式不得是值不变的
Rule 14.4	必要	if 语句和循环语句的控制表达式的基本类型应为布尔型

控制流(Control flow)

Rule 15.1	建议	不应使用 goto 语句
Rule 15.2	必要	goto 语句仅允许跳到在同一函数中声明的稍后位置的标签
Rule 15.3	必要	goto 语句引用的标签必须在goto 语句所在代码块或包含该代码块的上级代码块中声明
Rule 15.4	建议	最多只能有一个用于终止循环语句的 break 或 goto 语句
Rule 15.5	建议	应仅在函数的末尾有单个函数出口
Rule 15.6	必要	循环语句和选择语句的主体应为复合语句
Rule 15.7	必要	所有的 if…else if 构造都应以 else 语句结束

Switch 语句(Switch statements)

Rule 16.1	必要	switch 语句应格式正确
Rule 16.2	必要	switch 标签只能出现在构成 switch 语句主体的复合语句的最外层
Rule 16.3	必要	每一个 switch 子句(switch-clause)都应以无条件 break 语句终止
Rule 16.4	必要	每个 switch 语句都应具有 default 标签
Rule 16.5	必要	Default 标签应作为 switch 语句的第一个或最后一个 switch 标签
Rule 16.6	必要	每个 switch 语句应至少有两个switch 子句
Rule 16.7	必要	switch 语句的控制表达式(switch-expression)的基本类型不得是布尔型

函数(Functions)

Rule 17.1	必要	不得使用<stdarg.h>的功能
Rule 17.2	必要	函数不得直接或间接调用自身(不得使用递归函数)
Rule 17.3	强制	禁止隐式声明函数
Rule 17.4	强制	具有非 void 返回类型的函数的所有退出路径都应为具有带有表达式的显式 return 语句
Rule 17.5	建议	与数组型函数形参对应的函数入参应具有适当数量的元素
Rule 17.6	强制	数组形参的声明不得在[]之间包含 static 关键字
Rule 17.7	必要	非 void 返回类型的函数的返回值应该被使用
Rule 17.8	建议	不应更改函数形参

指针和数组(Pointers and arrays)

Rule 18.1	必要	指针操作数的算术运算应仅用于寻址与该指针操作数相同数组的元素
Rule 18.2	必要	指针之间的减法应仅用于寻址同一数组元素的指针
Rule 18.3	必要	关系运算符>，> =，<和<=不得应用于指针类型的对象，除非它们指向同一对象
Rule 18.4	建议	+，-，+=和-=运算符不得应用于指针类型的表达式
Rule 18.5	建议	声明中最多包含两层指针嵌套
Rule 18.6	必要	具有自动存储功能的对象的地址不得复制给在它的生命周期结束后仍会存在的另一个对象
Rule 18.7	必要	不得声明灵活数组成员
Rule 18.8	必要	不得使用可变长数组类型

重叠存储(Overlapping storage)

Rule 19.1	强制	不得将对象赋值或复制给重叠的对象
Rule 19.2	必要	不得使用 union 关键字

预处理指令(Preprocessing directives)

Rule 20.1	建议	#include 指令之前仅允许出现预处理指令或注释
Rule 20.2	必要	头文件名中不得出现“’”、“"”、“\”、字符以及“/*”或“//”字符序列
Rule 20.3	必要	#include 指令后须跟随<filename>或"filename"序列
Rule 20.4	必要	宏不得与关键字同名

Rule 20.5	建议	不应使用#undef
Rule 20.6	必要	看起来像预处理指令的符号不得出现在宏参数内
Rule 20.7	必要	宏参数展开产生的表达式应放在括号内
Rule 20.8	必要	#if 或#elif 预处理指令的控制表达式的计算结果应为 0 或 1
Rule 20.9	必要	#if 或#elif 预处理指令的控制表达式中使用的所有标识符应在其评估前被#define 定义
Rule 20.10	建议	不应使用“#”和“##”预处理运算符
Rule 20.11	必要	紧跟在“#”运算符之后的宏参数后面不得紧随“##”运算符
Rule 20.12	必要	用作“#”或“##”运算符的操作数的宏参数，不得是本身需要进一步宏替换的操作数
Rule 20.13	必要	以“#”作为第一个字符的一行代码应为有效的预处理指令
Rule 20.14	必要	所有#else，#elif 和#endif 预处理程序指令都应和与其相关的#if，#ifdef 或#ifndef 指令位于同一文件中

标准库(Standard libraries)

Rule 21.1	必要	不得将#define 和#undef 用于保留的标识符或保留的宏名称
Rule 21.2	必要	不得声明保留的标识符或宏名称
Rule 21.3	必要	不得使用<stdlib.h>中的内存分配和释放函数
Rule 21.4	必要	不得使用标准头文件<setjmp.h>
Rule 21.5	必要	不得使用标准头文件<signal.h>
Rule 21.6	必要	不得使用标准库输入/输出函数
Rule 21.7	必要	不得使用<stdlib.h>中的 atof、atoi、atol 和 atoll 函数
Rule 21.8	必要	不得使用<stdlib.h>中的 abort, exit, getenv 和 system 函数
Rule 21.9	必要	不得使用<stdlib.h>中的 bsearch 和 qsort 函数
Rule 21.10	必要	不得使用标准库时间和日期功能
Rule 21.11	必要	不得使用标准头文件<tgmath.h>
Rule 21.12	建议	不得使用<fenv.h>的异常处理功能

资源(Resources)

Rule 22.1	必要	通过标准库功能动态获取的所有资源均应明确释放

Rule 22.2	强制	只有通过标准库函数分配的内存块才能释放
Rule 22.3	必要	不得在不同的数据流上同时打开同一文件以进行读写访问
Rule 22.4	强制	禁止尝试对以只读方式打开的流执行写操作
Rule 22.5	强制	禁止反引用指向 FILE 对象的指针
Rule 22.6	强制	关联的流关闭后，禁止再使用指向 FILE 的指针值

附录 B：准则属性(Guideline attributes)

准则	级别	适用	分析
Dir 1.1	必要	C90，C99	
Dir 2.1	必要	C90，C99	
Dir 3.1	必要	C90，C99	
Dir 4.1	必要	C90，C99	
Dir 4.2	建议	C90，C99	
Dir 4.3	必要	C90，C99	
Dir 4.4	建议	C90，C99	
Dir 4.5	建议	C90，C99	
Dir 4.6	建议	C90，C99	
Dir 4.7	必要	C90，C99	
Dir 4.8	建议	C90，C99	
Dir 4.9	建议	C90，C99	
Dir 4.10	必要	C90，C99	
Dir 4.11	必要	C90，C99	
Dir 4.12	必要	C90，C99	
Dir 4.13	建议	C90，C99	
Rule 1.1	必要	C90，C99	可判定，单一编译单元
Rule 1.2	建议	C90，C99	不可判定，单一编译单元
Rule 1.3	必要	C90，C99	不可判定，系统范围
Rule 2.1	必要	C90，C99	不可判定，系统范围
Rule 2.2	必要	C90，C99	不可判定，系统范围
Rule 2.3	建议	C90，C99	可判定，系统范围
Rule 2.4	建议	C90，C99	可判定，系统范围
Rule 2.5	建议	C90，C99	可判定，系统范围
Rule 2.6	建议	C90，C99	可判定，单一编译单元
Rule 2.7	建议	C90，C99	可判定，单一编译单元

Rule 3.1	必要	C90，C99	可判定，单一编译单元
Rule 3.2	必要	C99	可判定，单一编译单元
Rule 4.1	必要	C90，C99	可判定，单一编译单元
Rule 4.2	建议	C90，C99	可判定，单一编译单元
Rule 5.1	必要	C90，C99	可判定，系统范围
Rule 5.2	必要	C90，C99	可判定，单一编译单元
Rule 5.3	必要	C90，C99	可判定，单一编译单元
Rule 5.4	必要	C90，C99	可判定，单一编译单元
Rule 5.5	必要	C90，C99	可判定，单一编译单元
Rule 5.6	必要	C90，C99	可判定，系统范围
Rule 5.7	必要	C90，C99	可判定，系统范围
Rule 5.8	必要	C90，C99	可判定，系统范围
Rule 5.9	建议	C90，C99	可判定，系统范围
Rule 6.1	必要	C90，C99	可判定，单一编译单元
Rule 6.2	必要	C90，C99	可判定，单一编译单元
Rule 7.1	必要	C90，C99	可判定，单一编译单元
Rule 7.2	必要	C90，C99	可判定，单一编译单元
Rule 7.3	必要	C90，C99	可判定，单一编译单元
Rule 7.4	必要	C90，C99	可判定，单一编译单元
Rule 8.1	必要	C90	可判定，单一编译单元
Rule 8.2	必要	C90，C99	可判定，单一编译单元
Rule 8.3	必要	C90，C99	可判定，系统范围
Rule 8.4	必要	C90，C99	可判定，单一编译单元
Rule 8.5	必要	C90，C99	可判定，系统范围
Rule 8.6	必要	C90，C99	可判定，系统范围
Rule 8.7	建议	C90，C99	可判定，系统范围
Rule 8.8	必要	C90，C99	可判定，单一编译单元
Rule 8.9	建议	C90，C99	可判定，系统范围
Rule 8.10	必要	C99	可判定，单一编译单元

Rule 8.11	建议	C90，C99	可判定，单一编译单元
Rule 8.12	必要	C90，C99	可判定，单一编译单元
Rule 8.13	建议	C90，C99	不可判定，系统范围
Rule 8.14	必要	C99	可判定，单一编译单元
Rule 9.1	强制	C90，C99	不可判定，系统范围
Rule 9.2	必要	C90，C99	可判定，单一编译单元
Rule 9.3	必要	C90，C99	可判定，单一编译单元
Rule 9.4	必要	C99	可判定，单一编译单元
Rule 9.5	必要	C99	可判定，单一编译单元
Rule 10.1	必要	C90，C99	可判定，单一编译单元
Rule 10.2	必要	C90，C99	可判定，单一编译单元
Rule 10.3	必要	C90，C99	可判定，单一编译单元
Rule 10.4	必要	C90，C99	可判定，单一编译单元
Rule 10.5	建议	C90，C99	可判定，单一编译单元
Rule 10.6	必要	C90，C99	可判定，单一编译单元
Rule 10.7	必要	C90，C99	可判定，单一编译单元
Rule 10.8	必要	C90，C99	可判定，单一编译单元
Rule 11.1	必要	C90，C99	可判定，单一编译单元
Rule 11.2	必要	C90，C99	可判定，单一编译单元
Rule 11.3	必要	C90，C99	可判定，单一编译单元
Rule 11.4	建议	C90，C99	可判定，单一编译单元
Rule 11.5	建议	C90，C99	可判定，单一编译单元
Rule 11.6	必要	C90，C99	可判定，单一编译单元
Rule 11.7	必要	C90，C99	可判定，单一编译单元
Rule 11.8	必要	C90，C99	可判定，单一编译单元
Rule 11.9	必要	C90，C99	可判定，单一编译单元
Rule 12.1	建议	C90，C99	可判定，单一编译单元
Rule 12.2	必要	C90，C99	不可判定，系统范围
Rule 12.3	建议	C90，C99	可判定，单一编译单元

Rule 12.4	建议	C90，C99	可判定，单一编译单元
Rule 13.1	必要	C99	不可判定，系统范围
Rule 13.2	必要	C90，C99	不可判定，系统范围
Rule 13.3	建议	C90，C99	可判定，单一编译单元
Rule 13.4	建议	C90，C99	可判定，单一编译单元
Rule 13.5	必要	C90，C99	不可判定，系统范围
Rule 13.6	强制	C90，C99	可判定，单一编译单元
Rule 14.1	必要	C90，C99	不可判定，系统范围
Rule 14.2	必要	C90，C99	不可判定，系统范围
Rule 14.3	必要	C90，C99	不可判定，系统范围
Rule 14.4	必要	C90，C99	可判定，单一编译单元
Rule 15.1	建议	C90，C99	可判定，单一编译单元
Rule 15.2	必要	C90，C99	可判定，单一编译单元
Rule 15.3	必要	C90，C99	可判定，单一编译单元
Rule 15.4	建议	C90，C99	可判定，单一编译单元
Rule 15.5	建议	C90，C99	可判定，单一编译单元
Rule 15.6	必要	C90，C99	可判定，单一编译单元
Rule 15.7	必要	C90，C99	可判定，单一编译单元
Rule 16.1	必要	C90，C99	可判定，单一编译单元
Rule 16.2	必要	C90，C99	可判定，单一编译单元
Rule 16.3	必要	C90，C99	可判定，单一编译单元
Rule 16.4	必要	C90，C99	可判定，单一编译单元
Rule 16.5	必要	C90，C99	可判定，单一编译单元
Rule 16.6	必要	C90，C99	可判定，单一编译单元
Rule 16.7	必要	C90，C99	可判定，单一编译单元
Rule 17.1	必要	C90，C99	可判定，单一编译单元
Rule 17.2	必要	C90	不可判定，系统范围
Rule 17.3	强制	C90，C99	可判定，单一编译单元
Rule 17.4	强制	C90，C99	可判定，单一编译单元

Rule 17.5	建议	C90，C99	不可判定，系统范围
Rule 17.6	强制	C99	可判定，单一编译单元
Rule 17.7	必要	C90，C99	不可判定，系统范围
Rule 17.8	建议	C90，C99	可判定，单一编译单元
Rule 18.1	必要	C90，C99	不可判定，系统范围
Rule 18.2	必要	C90，C99	不可判定，系统范围
Rule 18.3	必要	C90，C99	不可判定，系统范围
Rule 18.4	建议	C90，C99	可判定，单一编译单元
Rule 18.5	建议	C90，C99	可判定，单一编译单元
Rule 18.6	必要	C90，C99	不可判定，系统范围
Rule 18.7	必要	C99	可判定，单一编译单元
Rule 18.8	必要	C99	可判定，单一编译单元
Rule 19.1	强制	C90，C99	不可判定，系统范围
Rule 19.2	必要	C90，C99	可判定，单一编译单元
Rule 20.1	建议	C90，C99	可判定，单一编译单元
Rule 20.2	必要	C90，C99	可判定，单一编译单元
Rule 20.3	必要	C90，C99	可判定，单一编译单元
Rule 20.4	必要	C90，C99	可判定，单一编译单元
Rule 20.5	建议	C90，C99	可判定，单一编译单元
Rule 20.6	必要	C90，C99	可判定，单一编译单元
Rule 20.7	必要	C90，C99	可判定，单一编译单元
Rule 20.8	必要	C90，C99	可判定，单一编译单元
Rule 20.9	必要	C90，C99	可判定，单一编译单元
Rule 20.10	建议	C90，C99	可判定，单一编译单元
Rule 20.11	必要	C90，C99	可判定，单一编译单元
Rule 20.12	必要	C90，C99	可判定，单一编译单元
Rule 20.13	必要	C90，C99	可判定，单一编译单元
Rule 20.14	必要	C90，C99	可判定，单一编译单元
Rule 21.1	必要	C90，C99	可判定，单一编译单元

Rule 21.2	必要	C90，C99	可判定，单一编译单元
Rule 21.3	必要	C90，C99	可判定，单一编译单元
Rule 21.4	必要	C90，C99	可判定，单一编译单元
Rule 21.5	必要	C90，C99	可判定，单一编译单元
Rule 21.6	必要	C90，C99	可判定，单一编译单元
Rule 21.7	必要	C90，C99	可判定，单一编译单元
Rule 21.8	必要	C90，C99	可判定，单一编译单元
Rule 21.9	必要	C90，C99	可判定，单一编译单元
Rule 21.10	必要	C90，C99	可判定，单一编译单元
Rule 21.11	必要	C99	可判定，单一编译单元
Rule 21.12	建议	C99	可判定，单一编译单元
Rule 22.1	必要	C90，C99	不可判定，系统范围
Rule 22.2	强制	C90，C99	不可判定，系统范围
Rule 22.3	必要	C90，C99	不可判定，系统范围
Rule 22.4	强制	C90，C99	不可判定，系统范围
Rule 22.5	强制	C90，C99	不可判定，系统范围
Rule 22.6	强制	C90，C99	不可判定，系统范围

附录 C：C 语言类型安全(Type safety issues with C)

ISO C 被认为具有较差的类型安全性，因为它允许进行广泛的隐式类型转换。这些类型转换可能会危害安全性，因为其实现定义的方面可能会引起开发人员的困惑。

接下来是对这些转换的解释，并提供了一些示例来说明它们如何导致开发人员困惑。

C.1 类型转换(Type conversions)

C.1.1 隐式转换

ISO C 中的隐式类型转换允许在表达式中使用类型混合，而且即使所使用的类型都相同，也有可能发生。隐式转换分为三种类型：

1. 按 C90 第 6.2.1.1 节，C99 节第 6.3.1.1 节中所述，每当使用具有(un)signed char，(un)signed short 或 enum 类型的位域或对象进行算数运算时，位域或对象都会进行整数提升转换为 signed int 或 unsigned int。整数提升保留了原始值，但其符号位可能会更改(例如，unsigned char 将被转换为 signed int)；
2. 按 C90 第 6.2.1.5 节，C99 第 6.3.1.8 节所述，每当某些运算符使用不同类型的操作数时，通常的算术转换会将操作数转换(平级调整)为通用类型；
3. 赋值时转换。

表达式可以包含零个，一个，两个或所有这些隐式转换，如下面的示例所示(si 和 ui 是 32 位有符号和无符号整数，而 u8 是 8 位 unsigned char 型)：

表达式	整数提升	平级调整	转换	描述
si = si + si;				
si = uc + uc;	是			uc 整数提升为 signed int
ui = si + ui;		是		si 平级调整为 unsigned int
ui = ui + uc;	是	是		uc 整数提升为 signed int ，然后平级调整为 unsigned int
ui = si;			是	si 转换为 unsigned int
ui = uc + uc;	是		是	uc 整数提升为 signed ， 表达式结果转换为

				unsigned int
si = si + ui;		是	是	si 平级调整为 unsigned， 表达式结果转换为 signed int
si = ui + uc;	是	是	是	uc 整数提升为 signed int ，然后平级调整为 unsigned，最后表达式结果转换为 signed int

C.1.2 显式转换

出于功能原因，可能会引入显式(强制)类型转换：

- 更改执行后续算术运算的类型；
- 故意截断值；
- 为了清楚起见，使类型转换显式。

ISO C 不需要检查所有显式的强制类型转换，允许在不兼容的类型之间进行转换。

C.1.3 与转换有关的问题

隐式和显式转换可能导致一些问题，包括：

- 值损失：例如，转换为无法表示值大小的类型；
- 符号丢失：例如，从有符号类型转换为无符号类型会导致符号丢失；
- 精度损失：例如，从浮点型到整数型的转换，从而导致精度损失；
- 布局丢失：例如，从一种类型的指针到另一种类型的指针的转换导致存储布局不兼容。

虽然许多类型转换是安全的，但是可以保证所有数据值和所有可能的符合实现的安全的唯二转换是：

- 将整数值转换为具有相同符号的更广泛的整数类型；
- 将浮点类型转换为较宽的浮点类型。

取决于实现的整数大小，其他类型转换也可能是安全的。但是，任何潜在的危险类型转换都应通过显示标识将其明确表示出来。

C.2 开发人员困惑(Developer confusion)

由于实现类型的大小在不同的实现之间可能有所不同，因此进行隐式转换的上下文也会有所不同。 这就要求开发人员维护用于任何特定项目的类型系统的思维模型。 对于开发人员而言，不时地使用错误的模型是很容易发生的，特别是当他们处理具有不同实现的多个项目时。

C.2.1 整数提升中的类型扩展

计算整数表达式的类型取决于所有整数提升后的操作数的类型。将两个 8 位值相乘将始终得到至少 16 位宽的结果。但是，将两个 16 位值相乘只会在 int 的实现大小至少为 32 位时才会给出 32 位结果。永远不要依赖整数提升获得的类型扩展，因为相关的实现定义的行为可能会导致开发人员困惑。 考虑以下示例：

```
uint16_t u16a = 40000;  /* unsigned short or unsigned int ? */
uint16_t u16b = 30000;  /* unsigned short or unsigned int ? */
uint32_t u32x;          /* unsigned int or unsigned long ?  */

u32x = u16a + u16b;     /* u32x = 70000 or 4464 ?           */
```

预期的结果可能是 70000，但是分配给 u32x 的值将取决于 int 的实现大小。如果 int 是 32 位，则将以 32 位有符号算术进行加法，并且将获得“正确”值。如果只有 16 位，则将以 16 位无符号算术进行加法，将发生环绕并且将产生值 4464(70000％65536)。无符号算术的环绕式定义很明确，但由于其效果取决于执行该算术的类型，因此其有意使用应当被记录。

C.2.2 类型评估混乱

开发人员被欺骗，误以为进行计算的类型受分配或转换结果的类型影响的情况很容易发生。例如，在以下代码中，两个 16 位对象以 16 位算术加在一起(假定 int 为 16 位)，并且在赋值时将结果转换为 uint32_t 类型：

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

由于 u32x 的类型，开发人员通常会误以为加法是在 32 位算术上执行的。

这种性质的混淆不限于整数算术或隐式转换。下面的示例演示了一些语句，在这些语句中可以很好地定义结果，但是可能无法按照开发人员期望的类型执行计算：

```
u32a = (uint32_t)(u16a * u16b);     /* 可能不是32位运算          */
f64a = u16a / u16b;                 /* 非浮点数除法              */
f32a = (float32_t)(u16a / u16b);    /* 非浮点数除法              */
f64a = f32a + f32b;                 /* 可能是低精度较低的运算    */
f64a = (float64_t)(f32a + f32b);    /* 可能是低精度较低的运算    */
```

C.2.3 算术运算中符号的变化

整数提升可能会导致表达不符合开发人员期望的结果。 例如，表达式 10-u16a 产生的结果取决于 int 的大小。如果 u16a 的值为 100，并且 int 为 32 位，则结果将是值为-90 的有符号数。 但是，如果 int 为 16 位，则结果将为值为 65446 的无符号数。

C.2.4 按位运算中符号的变化

将按位运算符应用于小型无符号类型时出现的整数提升可能导致混乱。例如，对 unsigned char 型的操作数进行按位补码运算通常会产生带负值的(有符号)int 类型的结果。在操作之前，操作数将被提升为 int 类型，并且补码过程会设置额外的高阶位。多余的位数(如果有)取决于 int 的大小，如果补码运算后紧跟右移，则这特别危险，因为这会导致实现定义的行为。

```
u8a = 0xff;
if (~u8a == 0x00U)  /* This test will always fail */
```

附录 D：基本类型(Essential types)

ISO C 标准定义的 C 语言包含许多弱点和不一致之处，而且，不幸的是，表达式的标准类型也并不总是完全描述所表示数据的本质。 例如：

- **整数提升**：整数类型集的行为方式不一致。 作为整数提升的结果，基本类型为 unsigned 的数据组成的表达式(例如，unsigned char + unsigned char)的标准类型通常为 signed int。
- **整数常量**：整数常量仅适用于有符号/无符号的 int，long 和 long long 类型。这使确保类型一致性的问题变得复杂(例如，在将常量作为函数参数传递时)。
- **字符常量**：字符常量的标准类型(例如'x')定义为 int(而不是 char)。这模糊了字符数据(即“characters”)和数字数据(即“number values”)之间的区别。
- **逻辑表达式**：C90 语言中不存在布尔类型。 C99 中引入了_Bool 类型，但是不幸的是，由于向后兼容的原因，等于(==，!=)，关系(<，<=，>=，>)和逻辑(&&，||，!)运算符的类型仍然是 int 而不是_Bool 类型。
- **位域**：ISO C 没有定义位域类型。 这意味着，例如，无法转换为位域类型。 ISO C 定义的位域的标准类型是_Bool，signed int 或 unsigned int 之一。 类型为 int 的位字段的符号是实现定义的。
- **枚举**：枚举具有实现定义的类型。该实现可以自由使用任何能够表示枚举的整数类型。如果枚举包括负值，则该类型将为有符号整数类型。 如果枚举仅由非负值组成，则该类型可以是有符号或无符号整数类型。
- **枚举常量**：不管用于实现枚举的类型如何，枚举常量始终为 int 类型。

D.1 表达式的基本类型分类(The essential type category of expressions)

MISRA C：2012 引入了基本类型的概念，以帮助缓解上述问题。表达式的基本类型分类为：

- 基本型：布尔值；
- 基本型：字符；
- 基本型：枚举；
- 基本型：有符号(整型数)；
- 基本型：无符号(整型数)；
- 基本型：浮点型。

下表显示了标准整数类型如何映射到基本类型类别。 注意：C99 实现可以提供扩展的整数类型，每种扩展类型将被分配一个与其等级和符号相符的位置(请参阅 C99 第6.3.1.1 节)。

基本类型分类					
布尔型	字符型	有符号型	无符号型	enum <i>	浮点型
_Bool	char	signed char signed short signed int signed long signed long long	unsigned char unsigned short unsigned int unsigned long unsigned long long	named enum (命名枚举)	float double long double

在描述有符号和无符号类型的集合时，等级(rank)的概念尤其重要。有符号和无符号的 long long 的等级最高，而有符号和无符号的 char 的等级最低。在 ISO C99 标准中，“等级(rank)”是仅适用于整数类型的术语。

表达式的基本类型仅在标准类型为 signed int 或 unsigned int 的表达式中与标准 C 类型(标准类型)不同。

在以下各段中，定义了基本类型和标准类型不同的特定情况。

D.2 字符数据的基本类型(The essential type of character data)

具有标准 char 类型(即单字节字符)的对象基本类型为 char 型。

D.3 最低等级的有符号和无符号类型(STLR 和 UTLR)(The signed and unsigned type of lowest rank (STLR and UTLR))

基本类型模型引入了最低等级的有符号类型(Signed Type of Lowest Rank，STLR)和最低等级的无符号类型(Unsigned Type of Lowest Rank，UTLR)的概念。这些允许整数常量表达式和位域在具有低于 int 等级的基本类型的表达式中使用。对于整数常量表达式，这是一种方便，因为它避免了将表达式或其操作数转换为具有较低等级的类型的需要。类似地，对于不允许位域具有等级低于 int 等级的类型的实现，它避免了在某些情况下强制转换位域的需要。

1. STLR 是带符号的类型，具有表示特定整数常量表达式的值或位域的最大值和最小值所需的最低等级；

2. UTLR 是无符号类型，具有表示特定整数常量表达式的值或位域的最大值所需的最低等级。

D.4 位域的基本类型(The essenti al type of bit-fields)

位域的基本类型由以下适用的第一种确定：

1. 对于用基本布尔类型实现的位域，它的基本类型为布尔型；
2. 对于用带符号类型实现的位域，可以表示该位域的是 STLR。
3. 对于用无符号类型实现的位域，可以表示该是 UTLR。

D.5 枚举的基本类型(The essenti al type of enumerations)

枚举类型分两种：

1. 命名枚举类型是具有标签或被使用于任何对象，函数或类型的定义的枚举；
2. 匿名枚举类型是没有标签且未在任何对象，函数或类型的定义中使用的枚举。它通常将用于定义一组常量，这些常量可能相关也可能不相关。

一个命名枚举的类型与所有其他具名枚的举类型不同，即使它们在内部范围中使用完全相同的标记和枚举常量声明。命名枚举类型的每个实例在本文档中均表示为 enum <i>，其中每个 i 均不同。

使用 typedef 创建的命名枚举类型的别名仍表示该命名枚举类型，而不是新类型。

匿名枚举类型的基本类型是其标准类型。

以下所有都为命名枚举类型，并且具有不同的基本类型：

```
        enum ETAG { A, B, C };
typedef enum      { A, B, C } ETYPE;
typedef enum ETAG { A, B, C } ETYPE;
        enum      { A, B, C } x;
```

以下 typedef 定义的类型表示相同的 enum <i>类型：

```
typedef enum { A, B, C } ETYPE;
typedef ETYPE            FTYPE;
```

D.6 文字常量的基本类型(The essential type of literal constants)

ISO C99 标准定义了以下整数类型的常量：

- 整数常量；
- 枚举常量；
- 字符常量。

小贴士：整数类型的常量不一定是整数常量。

整数常量

1.如果整数常量的标准类型为 int，则其基本类型为 STLR；

2.如果整数常量的标准类型为 unsigned int，则其基本类型为 UTLR。

枚举常量

C 语言中的枚举常量的标准类型始终为 int，无论枚举的实现类型如何，也无论用于定义其值的任何初始化程序表达式的类型如何。例如，即使枚举常量使用无符号值(例如 500U)初始化，该常量仍将被视为具有标准类型的 signed int。

枚举常量的基本类型确定如下：

1.如果枚举定义了命名的枚举类型，则其枚举常量的基本类型为 enum <i>；

1.1 如果使用命名枚举类型定义基本布尔类型，则其枚举常量的基本类型为布尔型。2.

如果枚举定义了匿名枚举类型，则每个枚举常量的基本类型是其值的 STLR。

在下面示例里的每个枚举常量和对象 x 都具有 enum <i>的基本类型：

```
enum ETAG { A = 8, B = 64, C = 128 } x;
```

以下匿名枚举类型定义了一组常量。 每个常量的基本类型是 STLR。 因此，在具有 8 位 char 和 16 位 short 类型的机器上，A 和 B 基本类型为 signed char，而 C 的基本类型为signed short。

```
enum { A = 8, B = 64, C = 128 };
```

字符常量

字符常量的标准类型(例如'q'，'xy')是 int，而不是 char。

1.如果一个字符常量由单个字符组成，则其基本类型为 char;

2.其他基本类型与其标准类型相同。

布尔常量

C99 标准没有提供语法来显式定义_Bool 类型的常量。 库头文件<stdbool.h>根据 int 类型的常量 0 和1定义了 false 和 true，但是它们的基本类型仍定义为布尔类型。如果它们未被使用或不可见，则可以使用显式强制转换或提供基本布尔值的运算符声明类型为_Bool 的常量表达式。

工具可以提供很多其他方法，以识别基本布尔类型。例如，可以将工具配置为识别：

```
enum Bool {False，True};
```

在预处理指令中使用强制类型转换是一种语法错误。所以，仅当不打算在#if 或#elif 预处理指令中使用该表达式时，才能使用强制转换来定义布尔类型(例如(_Bool)0)的常量表达式。

D.7 表达式的基本类型(The essential type of expressions)

本节中未列出的表达式的基本类型与其标准类型相同。

逗号(,)

结果的基本类型是其右操作数的基本类型。

关系运算(<、<=、>=、>)、相等运算(==、!=)、逻辑运算(&&、||)

表达式的结果的基本类型为布尔型。

移位运算(<<、>>)

1.如果左操作数基本类型为无符号型，则：

1.1 如果两个操作数都是整数常量表达式，则结果的基本类型为结果值的 UTLR；

1.2 其他情况，结果的基本类型是左操作数的基本类型。

2.其他情况，基本类型是标准类型。

按位取反/取补码(~)

1.如果操作数基本类型为无符号型，则：

1.1 如果操作数是整数常量表达式，则结果的基本类型为结果值的 UTLR；

1.2 其他情况，结果的基本类型是操作数的基本类型。

2.其他情况，基本类型是标准类型。

单目运算：正(+)

1.如果操作数基本类型为有符号型或无符号型，则结果的基本类型是操作数的基本类型；

2.其他情况，基本类型是标准类型。

单目运算：负(-)

1.如果操作数基本类型为有符号型，则：

1.1 如果表达式是整数常量表达式，那么结果的基本类型是整个表达式的 STLR；

1.2 其他情况，结果的基本类型是操作数的基本类型。

2.其他情况，基本类型是标准类型。

条件运算(?:)

1.如果第二和第三操作数的基本类型相同，则结果具有相同的基本类型；

2.如果第二和第三操作数基本类型都为有符号型，则结果的基本类型是等级(rank)最高的那个操作数的基本类型。

3.如果第二和第三操作数基本类型都为无符号型，结果的基本类型是等级(rank)最高的那个的基本类

型。

4.其他情况，基本类型是标准类型。

需进行常规算术转换的运算(* / % + - & | ^)

1.如果两个操作数基本类型都为有符号型，则：

1.1 如果表达式是整数常量表达式，则结果的基本类型为结果的 STLR；

1.2 否则，结果的基本类型是等级(rank)最高的操作数的基本类型。

2.如果两个操作数基本类型都为无符号型，则：

2.1 如果表达式是整数常量表达式，则结果的基本类型为结果的 UTLR；

2.2 否则，结果的基本类型是等级(rank)最高的操作数的基本类型。

3.其他情况，基本类型是标准类型。

字符加法

1.如果一个操作数基本类型为字符型，而另一个操作数基本类型为有符号型或无符号型，则结果的基本类型为 char(字符型)；

2.其他情况，基本类型是标准类型(即将字符当做数值处理)。

字符减法

1.如果第一个操作数基本类型为字符型，第二个操作数基本类型为有符号型或无符号型，则结果的基本类型为 char(字符型)；

2.其他情况，基本类型是标准类型(即将字符当做数值处理)。

附录 E：适用于自动生成的代码(Applicability to automatically generated code)

略

附录 F：流程和工具清单(Process and tools checklist)

略

附录 G：实现定义的行为清单(Implementation-defined behaviour checklist)

略

附录 H：未定义和严重的未指定行为(Undefined and critical unspecified behaviour)

略

附录 I：背离说明示例(Example deviation record)

项目	F10_BCM	背离 ID	R_00102
MISRA C 条目	Rule 10.6	状态	已批准
源文件	Tool: MMMC	范围	项目

提交者	E C Unwin	批准者	D B Stevens
	签名		*签名*
职位	软件团队负责人	职位	工程主任
日期	27-Jul-2012	日期	12-Aug-2012

I.1 摘要

MISRA C：2012 Rule 10.6 的基本原理是，它避免了开发人员在进行某些算术运算的类型方面可能造成的混淆。 不幸的是，由于该项目使用的是编译器的“功能”，因此在不导致严重的运行时性能下降的情况下，不可能使代码符合规则。 这样做的影响是，在任何情况下都无法满足软件的计时要求，并且存在很大的风险，即无法达到车辆的法定碳氢化合物排放目标。

I.2 详细说明

此项目利用了几个可变的时间步长积分器，它们累积了一个多精度的长期总和。 要积分的数量和积分时间步长均为 16 位无符号数量，需要相乘才能得到 32 位结果，然后将其累加。

由于此项目上使用的C 编译器以 32 位实现 int 类型，因此计算 32 位乘积的代码为：

```
extern uint16_t qty, time_step;
uint32_t prod = (uint32_t)qty * (uint32_t)time_step;
```

显然，即使乘法运算符的操作数是 32 位，乘积对于 32 位也不可能太大，因为两个操作数都从 16 位进行了零扩展。

按照我们的标准开发程序，所有对象模块都将通过最坏情况的执行时间分析工具，以确保每个功能都满足体系结构设计中指定的执行时间预算。 分析人员强调，为这些集成商生成的代码远远超出了为他们计划的预算。 调查显示，执行时间过长的原因是编译器正在生成对长乘法例程的“移位加法”样式的调用。这是令人惊讶的，因为处理器配备了 IMUL 指令，该指令能够将两个 16 位无符号整数相乘以在单个时钟周期内提供 32 位结果。 尽管乘法操作数是 32 位，但是编译器可以知道这些操作数的最高 16 位为 0，因此它应该能够选择 IMUL 指令。

对编译器进行的实验表明，如果乘法的操作数隐式地从 16 位转换为 32 位，它将选择 IMUL 指令，即：

```
uint32_t prod = qty * time_step;
```

这特别奇怪，因为 C 标准所描述的代码的行为与转换是隐式还是显式无关。 尽管无论将 IMUL 还是库调用用于乘法，程序都将产生相同的结果，但库调用需要执行 100 个周期的最坏情况。 编译器供应商已书面确认这是他们在这种情况下所期望的行为。

I.3 理由

由于这种类型的积分器在项目中的多个功能中使用，并且平均至少每 100 微秒执行一次，因此该库功能的性能是不可接受的。 在指定的 CPU 内部频率 25 MHz 下，这意味着仅在这些乘法上花费了 4%的时间。这本身并不重要，并且可以包含在总体时序预算中可用的净空中。 但是，设计规定，积分器应在最多 10 微秒内提供其结果，以满足其他功能的时序要求。 未能满足此要求意味着实现排放目标存在巨大风险，其商业影响超过 1000 万美元。

我们对这个问题的首选解决方案是使用隐式转换编写积分器。对于此类集成商，这将需要违反MISRA C：2012 Rule 10.6。 该代码在功能上与由 MISRA 兼容的代码生成的代码相同，但是执行速度提高了 100 倍。

考虑了以下其他选项：

◆ 提高时钟速度-要获得所需的性能，需要将时钟增加 10 倍，但处理器的最大 PLL 频率为 100 MHz；

◆ 变更处理器-鉴于硬件设计验证正在进行中，因此在商业上不可行；该项目的额外费用约为 250000 美元，也将对时间产生影响；

◆ 更改编译器-此处理器没有其他商业认可的编译器；有不受支持的公共领域编译器，但不适用于该项目。

◆ 重新编码库例程—库使用以 2 为基的长乘法；可以使用 3 条 IMUL 指令对其进行重新编码，以实现以 65×536 为基数的长乘法，但是我们不愿意更改编译器供应商的代码；我们已征求他们对这种方法的看法，并收到答复说“他们无法支持我们对其图书馆进行更改”。

I.4 请求背离的场景

对于项目中可变时间步长积分器的所有实例，都要求此背离。

I.5 违规后果

在此背离记录中描述的情况下，不存在违反 MISRA C：2012 规则 10.6 的后果。

由于此背离，因此没有其他验证要求。

I.6 控制报告的措施

用于检查是否符合此规则的 MMMC 工具提供了一种工具，通过该工具可以在表达式中禁止显示诊断消息。由于所有积分器均为以下形式：

```
prod = qty * time_step;
```

宏可用于实现集成器并禁止显示警告。 以下宏将用于实现其结果与乘积项的乘法和赋值：

```
/* Violates Rule 10.6: See deviation R_00102 */
#define INTEG(prod, qty, time_step) \
( /* -mmmc-R10_6 */ (prod) = (qty) * (time_step) /* -mmmc-pop */ )
```

尽管可以将此宏实现为一个函数，但是鉴于所执行操作的简单性，调用和返回的开销过大。 因此，在这种情况下，宏优先于函数使用，虽然这意味着违反 Dir 4.9。

附录 J：词汇表(Glossary)

略