MISRA The Motor Industry Software Reliability Association

MISRA-C-:2004

Guidelines for the use of the C language in critical systems

中文版

1 背景 —— C 的使用和问题

1.1 汽车工业中 C 的使用

MISRA-C:1998 [1] 发布于 1998 年。本文档是它的修订版本，用来解决与第一版本有关系的问题。

在汽车工业领域的实时嵌入式应用中，C 编程语言的使用越来越体现出广泛性和重要性。这在相当程度上取决于该语言固有的灵活性、可支持的范围及其潜在的访问广泛硬件环境的可移植性。详细的理由包括：

- 对于许多使用中的微处理器来说，如果存在其他除了汇编语言之外的可用语言，通常就是 C。在许多情况下，其他语言根本就不可用于硬件。
- C 对高速、底层、输入/输出操作等提供了很好的支持，而这些特性是许多汽车嵌入式系统的基本特性。
- 由于应用的逐步增长的复杂性，高级语言的使用较汇编语言更为适合。
- 相对于其他一些高级语言，C 能够产生较小的和较少 RAM 密集性（RAM-intensive）的代码。
- 增长的可移植性需求。市场竞争要求在工程项目生命周期的任何阶段，软件可以通过移植到新的和/或低成本的处理器，目的是为了降低硬件成本。
- 增长的自动产生 C 代码的使用要求。C 代码需要从模型包中自动产生。
- 增长的对开放系统和主机环境（hosted enviroments）的兴趣。

1.2 语言的不安全性和 C 语言

没有哪种编程语言能够保证最终的可执行代码会准确地按照程序员预想的那样执行。任何语言都会产生大量的问题，下面为其做了广泛的分类，并描述了 C 语言不安全性的例子。

1.2.1 程序员产生错误

程序员产生的错误，简单的可以是变量名字的书写错误，或者更为复杂的错误，如对算法的误解。编程语言可以承受这样的错误。首先，语言的风格和表达能帮助或提示程序员清晰考虑其算法。其次，对于书写错误，语言可以使从一个有效结构向另一个有效（不是预想的）结构的转换变得轻松或困难。第三，当错误发生时，语言可以检测到也可能检测不到。

首先，关于语言的风格和表达，使用 C 可以编写出良好布局的、结构化的和表达性强的代码。还可以使用它编写出不正当的和特别难以理解的代码。很明显，后者对于安全相关的系统是不可接受的。

其次，C 的语法特性足以使得书写错误也能产生完全有效的代码。例如，在“＝＝”（逻辑比较）的地方写成“＝”（赋值）是很常见的，而且最终结果也几乎总是有效的（但它是错误的）；而 if 语句的结尾出现的多余分号能完全改变代码逻辑。

第三，C 的基本观点是假设程序员知道他们在做什么，这意味着错误即使出现也不会被语言注意到而通过。在这方面 C 体现出的软弱性正在于它的“书写检查”（type checking）。举例来说，C 不会拒绝程序员在使用整数代表 true/false 值时却在该整数中存储了浮点值。大多数这样的失配可以简单地通过强制使其合适。如果 C 的表现不得其所（a square peg and a round），

它不会挑剔而会适合它们！

1.2.2 程序员不了解语言

程序员可能会误解语言构造的作用。对这样的误解，一些语言是更为开放的。

C 语言中有相当多的地方能使程序员轻易产生误解。例如运算符优先级的规则。这些规则是良好定义的，但也非常复杂，也很容易对某特定表达式中运算符的优先级做出错误的假设。

1.2.3 编译器的行为同程序员预期的不同

如果语言具有未经完善定义的特性，或者模糊特性，那么在程序员认为某个构造应该如此时，编译器的解释却是完全不同的。

C 语言中许多地方是未经完善定义的，因此其行为可能会随着编译器的改变而改变。某些情况下，其行为甚至在同一个编译器内也会根据上下文而发生变化。Annex G 中所有的 C 标准列出了 201 个这种问题。这可以提出相当多的问题，特别是在面临编译器间的移植性时。然而，C 标准 [2] 列出了这些问题，所以它们能为我们所知。

1.2.4 编译器包含错误

语言的编译器（及其链接器等）本身就是软件工具。编译器可能不会始终正确地编译代码。例如，在特定环境下它们可能同语言标准相违背，或者其本身可能就包含“bug”。

因为 C 语言中存在许多难以理解的地方，编译器的编写者很容易错误地解释和实现标准。语言的某些地方比其他的更容易这样。而且编译器的编写者有时会有意改变标准。

1.2.5 运行时错误

有些不同的语言问题产生自正确编译的代码，但某些特殊数据会在代码运行时产生错误。语言能够对可执行代码内部做运行时的检查以检测这样的错误并执行适当的动作。

通常，C 的运行时检查能力比较弱。这也是 C 代码短小有效的原因之一，但是在运行中检查错误就要花费一定的代价。C 编译器通常不为某些常见问题提供运行时检查，诸如数学异常（如零除）、溢出、指针地址的有效性，或数组越界错误。

1.3 安全相关系统中 C 的使用

从 1.2 节可以清楚地知道，在安全相关系统中使用 C 语言需要相当小心。针对上面提及的问题种类，已经为安全相关系统中 C 的使用提出了许多需要关注的事情。显然地，不能为安全相关系统使用所有的 C 语言特性。

然而，做为语言来说，C 是非常成熟的，在实践中也是经过了良好分析和使用的。所以它的不足也是众所周知和可以理解的。同时可获得大量的商业工具支持，这些工具用来静态检查 C 源代码和提醒程序员语言问题的存在。

如果为了实践的原因有必要在安全相关系统中使用 C 语言，那么必须对语言的使用加以限制，避免那些确实可以产生问题的地方，直到它是可以应用的。本文档为人们提供了这样的限制集合（通常称为“语言子集”（language subset））。

Hatton [3] 认为，倘若强加了“……严格和自动的强制性约束……”，那么 C 能够写出“……至少同其他通用语言一样的具有高质量和一致性的软件”。

注意，相对于 C 语言，汇编语言不再适合于安全相关系统，在某些方面还会更坏。通常不建议在安全相关系统中使用汇编语言，即使使用也要加以非常严格的约束。

1.4 标准化

本文档使用的标准是由 ISO 9899:1990 [2] 定义的 C 编程语言，它是由 ISO/IEC 9899/COR1:1995 [4]、ISO/IEC 9899/AMD1:1995 [5]和 ISO/IEC 9899/COR2:1996 [6]修订的。为了简便，本文档中统称为“C90”。基本的 1990 年文档[7]是 ANSI X3.159-1989 [2]的 ISO 版本。在内容上，ISO/IEC 标准同 ANSI 标准是一致的。然而要说明的是，两个标准中章节的编号是不同的，本文档的章节编号随同 ISO 标准。

还要说明的是，ANSI 标准[7]包含了一个有用的附录，其中在某些由标准委员会制定的决策后给出了相关的说明解释。ISO 版本中没有这样的附录。

工作小组已经开始考虑 ISO/IEC 9899:1999[8]（称为“C99”）。在本文档的发布日期（2004 年 10 月），还没出现嵌入了 C99 的商业编译器。

2 MISRA-C：视野

2.1 MISRA-C 的发布说明

MISRA 协会在 1994 年发布了它的“Development Guidelines for Vehicle Based Software”[9]，描述了软件开发过程中所有应该使用的方法集。特别地，在使用中因为考虑安全集成度而做出的语言、编译器和语言特性的选择，成为首要考虑的事情。MISRA 指南[9]中的节 3.2.4.3（b）和表 3 描述了这些。推荐的方法之一是使用已经应用于航空、能源和国防工业中的标准化的语言子集。本文档定义了这样一个合适的 C 语言子集。

2.2 MISRA-C 的目标

在发布这个关于 C 编程语言使用的文档时，MISRA 协会并没想促进 C 在汽车工业中的应用，而是因为认识到了 C 的广泛应用。本文档的目标是为了促进该语言的最为安全的使用。

MISRA 协会希望本文档能获得工业上的接受，以及对更安全子集的采用能够成为汽车制造业和许多零部件供应商的最佳实践。它也应该为通用 C 和该特殊子集的使用增加培训和增强使用能力，无论是在个人层次还是公司范围。

大部分重点放在了静态检查工具上以强制对子集的适应，同时希望这也会成为汽车嵌入式系统开发者的普遍实践。

尽管许多学术著作对语言及其正反两面做了大量的描述，但这方面信息在汽车开发中还不是广为人知。本文档的另外一个目标就是使汽车工业中的工程师和经理们越来越多地关心语言的选用问题。

可以帮助进行软件开发的工具，尤其是支持 C 语言使用的工具对我们来说是个好处。然而，对设计与实现的鲁棒性的关注是始终存在的，特别是对安全相关软件的开发。人们希望汽车工业中正在使用的实现软件最佳实践（通过 MISRA 指南[9]和本文档）的方法能鼓励 COTS 工具的供应商使用它以保证其产品适合于汽车工业应用。

3 MISRA-C：范围

3.1 基本的语言问题

MISRA 指南[9]（表 3）要求使用“标准化结构化语言的受限子集”。对 C 来说，这意味着只能使用在 ISO 标准中定义的语言，因此排除了以下方面的使用：

- K&R C（由 Kernighan 和 Ritchie 编写的“The C Programming language”的第一版）
- C++
- C 的私有扩展

3.2 未指出的问题

风格和代码度量问题在一定程度上是很个人化的。对任何一群人来说都很难赞成什么是适当的，而要 MISRA 给出确定性的建议也是不合适的。重要的不是使用者采取了哪种确切的风格或者特定的度量，而是使用者要定义风格指南和合适的度量与限制（见节 4.2.2 和 4.2.4）。

MISRA 协会的角色不是建议特定的供应商或工具强制执行所采取的约束。使用本文档的用户在选择工具上是自由的，鼓励供应商提供遵循规则的工具。本文档用户的负担是要声明其工具充分地遵循了规则。

3.3 可应用性

本文档被设计为应用在汽车嵌入式系统的产品代码上。

根据 ISO 9899[2]（节 5.1.2）中定义的执行环境，本文档的目标定在“自立（free-standing）的环境”，尽管它也描述了库的问题，因为嵌入式编译器中经常也会用到一些标准库。

如果合适，本文档的大部分要求也可以应用在其他嵌入式领域。对主机系统（hosted system）的应用，本文档的要求不是必需的。

在进行编译器和静态工具的比较检测（benchmark）时也不需完全应用本文档规则，有时在比较工具时有必要打破这些规则，以测量工具的响应。

3.4 预备知识

本文档不是要做为所涉主题的介绍和培训。而是认为读者熟悉 ISO C 编程语言标准和相关工具，并读过了主要的引用文档。同样假设读者已经接受了适当的培训并胜任 C 程序员工作。

3.5 C++问题

C++不同于 C 语言，本文档的范围不包含它，也不试图讲解 C++对于编写安全相关系统的适合性与否。然而对于 C++编译器和代码的使用方面应该给出以下说明。

C++不仅仅是 C 的超集（即增加多余的 C 特性）。在 C 和 C++中有少数特殊构造具有不同的解释。而且有效的 C 代码可以包含一些在 C++中是保留字的标识符。基于这些原因，由 C

写成的符合 ISO 标准的代码在 C++编译器中可能不会编译成其在 C 编译器中的结果。因此本文档不赞成使用 C++编译器编译 C 代码。如果为了实用性原因必须使用 C++编译器编译 C 代码，那么首先必须完全理解两种语言的不同之处。

然而鼓励使用附加的编译器，如静态检查工具。基于此目的，在不关心可执行代码的地方，可以使用 C++编译器，而且这也确实能提供好处，因为它具有比 C 更强的类型检测能力。

如果一个做为“C++”销售的编译器严格地符合了 ISO C 的操作模式，那么它就相当于一个 C 编译器，并可以以 C 编译器的形式使用（只对于 C 代码）。对于其他任何包含了适应 ISO C 编译器的功能部分的工具而言，这也是成立的。

C++注释不应用在 C 代码中。尽管 C 编译器支持这种形式的注释（由//表示的），它们也不是 ISO 标准 C 的一部分（见规则 2.2）。

3.6 自动产生代码的问题

如果要使用代码自动产生工具，那么有必要选择一个合适的工具并采取有效性验证的手段。建议遵循本文档的要求，它可以提供评估工具的准则，除此之外在这方面没有更多的指南，读者可以参考 HSE 的 COTS 建议[10]。

为了进行有效性验证，对待自动产生代码的方式必须与对待手工产生代码的方式相同（见 MISRA 指南[9] 3.1.3，Planning for V&V）。

4 使用 MISRA-C

4.1 软件工程环境

使用编程语言产生源代码只是软件开发过程中的一个活动。如果只在这样的活动中坚持最佳实践而不考虑其他开发问题，只能带来有限的价值。在安全相关系统的开发中尤其如此。这些问题在 MISRA 指南[9]中都提到了，比如：

- 文档化开发过程
- 能够满足 ISO 9001/ISO 90003/Tick IT [11，12，13]标准的质量体系
- 项目管理
- 配置管理
- 风险分析（Hazard Analysis）
- 需求
- 设计
- 编码
- 确认（Verificaiton）
- 有效性验证（Validation）

软件开发者有必要证明他们整个的开发过程对其所开发的系统来说是合适的。这样的声明只有当执行完确定系统安全集成度的风险分析活动后才算完成。

4.2 编程语言和编码环境

在软件开发过程的编码阶段，语言子集只是许多方面之一，而如果不考虑其他开发问题，只在这一方面中坚持最佳实践，只能带来有限的价值。紧接语言的选择之后的，其他关键问题是：

- 培训
- 风格指南（Style guide）
- 编译器选择和有效性验证
- 检查工具的验证
- 度量
- 测试覆盖率

在这些问题上做出的所有选择及其原因都要记录成文档，并对任何执行的活动保持适当的记录。然后，在需要时这样的文档可以包含进安全性声明中。

4.2.1 培训（Training）

为了确保 C 代码编写者具有合适的熟练程度和能力，应该提供正规训练：

- 嵌入式应用中 C 编程语言的使用

- 高度集成的和安全相关的系统中 C 编程语言的使用
- 用于强制遵循子集的静态检查工具的使用

4.2.2 风格指南（Style guide）

除了采用语言子集外，一个组织应该有其自己的实验室内的编程风格。这包含那些不直接影响代码正确性但定义了代码呈现形式的“实验室风格”（house style）的问题。这些问题是主观的。风格指南涉及的典型问题包括：

- 代码布局和缩进的使用
- 大括号“{}”和结构块的布局
- 语句的复杂性
- 命名规范
- 注释语句的使用
- 公司名称、版权提示和其他标准文件头信息的包含

正如风格指南的某些内容是建议性的，某些内容也是强制性的。然而，对风格的强制不在本文档所讨论的范围之内。

关于风格的更多信息参见[14]。

4.2.3 工具选择和有效性验证

在选择编译器（可以理解为包括链接器）时，尽可能地要使用适应 ISO C 的编译器。如果语言的使用依赖于“实现定义”（implementation-defined）的特性（如 ISO 标准[2]的 Annex G.3 中所确认的），那么开发者必须检测编译器以确认其实现与编译器编写者所声明的一致。Annex G 的更多解释见 5.5.2 节。

在选择静态检查工具时，很明显地需要该工具尽可能多地强制遵循本文档的规则。为了这个目标，重要的是工具能在整个程序范围内进行检查，而不是单单一个源文件。另外，如果一个检查工具能执行超出本文档要求之外的检查，建议进行额外的检查。

编译器和静态检查工具通常被视作“可信的”（trusted）过程。这意味着对工具的输出具有相当程度的信任，因此程序员必须要确保这样的信任不是一种错信。理想的是，这种信任应该由工具的供应商通过运行合适的有效性测试来达到。需要说明的是，虽然能采用有效性验证的方法来为一个嵌入式目标测试编译器，但在本文档发布时还没有这样的形式化验证机制。另外，这些工具应该被开发成能够满足 ISO 9001/ISO 90003[11，12，13]的质量体系。

工具的供应商应该能提供确认和有效性验证活动的记录以及软件的可控开发过程的变更记录。供应商应该具有下面的机制：

- 记录由用户报告的缺陷
- 对现有客户通告已知的缺陷
- 在未来的发布中改正缺陷

现有客户群的规模以及对前 6 到 12 个月报告的缺陷的观察能够说明工具的稳定性。

通常不大可能从供应商那里得到这等层次的保证，在这些情况下开发者的职责是确保工具具有足够的质量。

开发者可以采用的增强工具信心的可能手段是：

- 执行某种形式的文档化验证测试
- 评估工具供应商的软件开发过程

- 检阅工具迄今为止的性能

可以通过创建代码例子在工具上运行来执行有效性验证。对编译器来说，它可以由前面应用中已知的良好代码组成。对静态检查工具，应该写一套代码文件，每个文件打破子集中的一条规则，并且所有文件在一起要覆盖尽可能多的规则。然后，静态检查工具为每个测试文件发现不适应的代码。尽管这样的测试是受限的，它们也将确立基本层次上的工具行为。

需要说明的是，在做编译器的有效性验证时，要使用与编译产品代码时相同的编译器选项、链接器选项和源文件库版本。

4.2.4 源代码复杂度

强烈建议使用源代码的复杂性度量。这可以防止编写出难以控制和难以测试的代码。同样高度建议使用工具进行数据的搜集工作。许多可以用于强制遵循子集的工具同样具有产生度量数据的能力。

详细的源代码度量参见“Software Metrics: A Rigorous and Practical Approach”（Fenton 和 Pfleeger）[15]，以及 MISRA 关于软件度量的报告[16]。

4.2.5 测试覆盖率

在软件设计和编写之前，应该定义好它期望达到的语句覆盖率。代码的设计和编写应该能使其在测试中达到很高的语句覆盖率。在机械、电气和电子领域，这个概念有个术语叫“Design For Test”（DFT）。在编写代码的过程中就应该考虑这个问题，因为达到高语句覆盖的能力是源代码的新兴属性。

使用子集，它减少了定义实现的属性的数量和增加了模块接口兼容性，能产生可以进行较大规模集成和测试的软件。

折中考虑以下的度量可以促进高语句覆盖率的实现：

- 代码规模
- 圈复杂度
- 静态路径数目

使用计划好的方法，在软件设计、语言使用和 DFT 上的花费可以由测试阶段中要达到高语句覆盖率所需时间的缩减来补偿。见[16，17]。

4.3 采用子集（subset）

为了开发符合子集要求的代码，需要遵循以下步骤：

- 产生一个符合性矩阵（compliance matrix），声明是如何强制每个规则的
- 产生一个背离（deviation）过程
- 在质量管理系统中统一工作实践的形式

4.3.1 符合性矩阵（Compliance matrix）

为了确保所编写的代码符合子集，有必要进行适当的测量，检查它没有打破规则。要做到这一点最有效的手段是，使用一个或多个静态检查工具。如果工具不能检查某条规则，那么就必须进行人工检查。

为了确保所有的规则都覆盖到了，就需要产生一个符合性矩阵，矩阵中列出每条规则并说明它是如何被检查的。例如表 1。见附录 A 中的规则列表，它可以帮助产生完整的符合性矩阵。

Rule No.	Comiler1	Compiler2	Checking Tool 1	Checking Tool 2	Manual Review
1.1	Warning 347				
1.2		Error 25			
1.3			Message 38		
1.4				Warning 97	
1.5					Proc x.y

Table 1: Example compliance matrix

如果开发者具有其他的本地约束，也可以加入到符合性矩阵中。在忽略特定约束的地方，要给出完整的说明。这些说明必须是能被 C 语言专家和管理者一同承认的。

4.3.2 背离过程（Deviation procedure）

要知道在某些情况下需要背离本文档给出的规则。例如，与微处理器硬件接口的源代码会不可避免地需要使用适当的语言扩展。

为了具有权威性，有必要使用一个正规的过程来认可这些背离，而不是依靠一个个体的程序员随意背离规则的判断力。背离的使用必须在必要性和安全性上被声明为是正确的。由于本文档不打算给出每条规则的重要等级，因此如果说某些条款是比其他条款更为关键也是可以接受的。这将反映在背离的过程中，如果对于更为严重的背离，需要更强的技术能力以评估即将招致的风险，需要更高的管理以接受这种增加的风险。如果存在正规化的质量管理系统，那么背离过程应该是该系统的一部分。

背离的发生可以是针对某一特殊实例，即在单一文件中一次性出现的事物，或者是针对某一类环境，即一特定环境中特定结构的系统级使用，如使用特定的语言扩展实现处理串口通讯的文件 I/O 操作。

严格遵循所有规则是不大可能的，而且在实际中，与个体情形有关的背离是可以容许的。背离的种类有两种：

- 项目背离（Project Deviation）：项目背离定义为可允许的对规则要求的放宽，以应用在特殊环境中。实际上，项目背离在项目开始时总是可以的。
- 特定背离（Specific Deviation）：特定背离定义为在某个文件中对某规则的背离，典型地由对环境的响应给出。特定背离发生在开发过程中。

项目背离需要正规地检查回顾，这样的检查应该是形式化背离过程的一部分。

一些需要打破规则的环境关心的是输入/输出的操作。建议软件的设计要使得对输入/输出的关注同软件的其他部分隔离开来。在代码的输入/输出部分中要尽可能地限制项目背离。服从于项目背离的代码应该标记出来。

本文档的目标是要仔细考虑这些问题并采取可靠的手段以避免这些问题。不应该使用背离过程去破坏这样的思想。在遵循本文档的规则时，开发者同时也利用了 MISRA 在理解这些问题时所做出的努力。如果要背离规则，那么开发者就必须要理解问题本身。所有的背离，包括标准的和特定的，都应该记录成文档。

例如，如果预先知道很难遵循某一规则，软件开发者应该提交一份项目背离请求（Project Deviation Request）并在开始编程之前征得客户方的同意。

项目背离请求应该包含如下内容：

- 背离的细节，即被背离的规则
- 需要产生背离的环境
- 由背离产生的潜在后果
- 对背离的正确性声明

- 声明如何保障安全性

如果在开发过程期间或结尾出现了对背离的需求，软件开发者应该提交一份特定背离请求（Specfic Deviation Request）。

特定背离请求应该包含如下内容：

- 背离的细节，即被背离的规则
- 由背离产生的潜在后果
- 对背离的正确性声明
- 声明如何保障安全性

这些过程的实现细节留给用户决定。

4.3.3 质量体系中的形式化

子集、静态检查工具和背离过程的使用应该由形式化的文档在质量管理体系中做出描述。然后它们要经过与质量体系相关的内部和外部审查，这将促进它们的一致性使用。

4.3.4 引入子集

如果一个组织已经具有 C 代码编程环境，那么建议引入本文档规则（见 Hatton[3]的第五章）。实现本文档的所有内容可能需要 1 到 2 年的时间。

如果某产品包含了在使用子集前编写的代码，重写这些代码以适应子集可能是不现实的。在这种情况下，开发者必须决定管理引入子集的策略（例如：所有新模块要写成符合子集的方式，而对于已存在的模块，如果它们要做出超过非注释语句 30%的更改，那么就要重写成符合子集的形式）。

4.4 符合性声明（Claiming compliance）

只能为产品而不能为组织声明符合性。

在为一个产品声明 MISRA-C 文档的符合性时，开发者的意思是在说，存在着体现下列内容的依据：

- 完成了符合性矩阵，它显示出如何强制符合性的
- 产品中所有 C 代码与本文档的规则是一致的，或者遵循了对文档规则的背离
- 维持了没有遵循的所有规则实例列表，对每个实例来说，……
- 4.2 节中提到的问题已经解决了

4.5 持续改进

对本文档规则的遵循只是持续改进过程的第一步。用户必须要知道其他主题内容（见参考），并且要主动使用度量寻求开发过程的改进。

5 规则简介

这一节描述文档第 6 节中规则的表达方式。是文档主要内容的简单介绍。

5.1 规则分类

第 6 节中的每条规则都被分类成“强制”（required）或“建议”（advisory），在后面描述。在这个基本分类之外，文档不打算给出每条规则的重要等级。所有强制的规则具有同等重要性，所有建议的规则也如此。文档中对某一项的忽略也不表明它的重要性较低。

“强制”和“建议”的含义如下：

5.1.1 强制规则

这是对程序员的强制要求。文档中共有 121 条“强制”规则。声明遵循本文档的 C 代码应该适合每条强制规则，如果不是，就需要具有形式化的背离，见 4.3.2 节。

5.1.2 建议规则

这些要求程序员在通常情况下都要遵守。然而它们不象强制规则那样带有强迫性质。文档中共有 20 条“建议”规则。要说明的是，“建议”不意味着可以忽略这些规则，而是应该遵守直至合理的实现。对建议规则来说不需要形式化的背离，但如果背离是适当的，也可以产生。

5.2 规则的组织

规则被组织在 C 语言的不同主题下。但不可避免地会有一些重叠，一条规则可能会同一些主题相关。这种情况下，规则被安排在最具相关性的主题下。

5.3 规则的冗余

本文档中存在少数这样的情况，某规则的给出针对某一语言特性，该语言特性在某处是禁止的而在他处却是建议使用的。这是有意的。用户可能要通过以下两种方式选用该语言特性，或者是产生对某强制规则的背离，或者是选择不遵循某条建议规则。这种情况下，第二条限制语言特性使用的规则成为相关的规则。

5.4 规则的形式

本文档中的个体规则书写形式如下：

规则 <序号> (<类别>): <规则文本>

[<原始参考>]

各内容如下：

- <序号>：每条规则带有唯一的序号。序号的前缀是规则的组别，后缀是在规则组中的序号。

- <类别>：或者是“强制”，或者是“建议”，见 5.1 节中的解释
- <规则文本>：规则内容
- <原始参考>：指示了产生本条款或本组条款的可应用的主要来源。5.5 节描述了这些参考的重要性以及到原始材料的连接。

另外，对每项条款或成组相关条款提供了支持文本。该文本描述了规则所涉及的基本问题及如何应用规则的例子。如果在某一规则后没有紧跟这样的解释文本，那么对应文本会在一组规则后找到，这段文本适应于其前所有规则。类似地，一组规则后的原始参考适用于整组规则。

支持文本不是做为相关语言特性的指南，我们假设读者已经具有关于语言的工作经验。语言特性的更为详细的信息可以通过咨询相关的语言标准或其他 C 语言参考书来获得。如果原始参考给出了一个或多个 ISO 标准中“Annex G”条款，那么 ISO 标准中提出的原始问题将有助于对规则的理解。

在代码段中，下列已经 typedef 定义的类型假设为（为了适应规则 6.3）：

char_t	plain 8 bit character
uint8_t	unsigned 8 bit integer
uin16_t	unsigned 16 bit integer
uint32_t	unsigned 32 bit integer
int8_t	signed 8 bit integer
int16_t	signed 16 bit integer
int32_t	signed 32 bit integer
float32_t	32 bit floating-point
float64_t	64 bit floating-point

非特意定义的变量名称指示其类型。例如：

uint8_t	u8a;
sint32_t	s32a;

5.5 理解原始参考

当规则来源于一个或多个已发表的原始文件时，这些来源会在一个方括号中标明。这样做有两个目的。首先，读者可以咨询这些特定的资源以充分了解规则之后的基本原理（比如需要对规则有所背离的时候）。其次，考虑到 ISO 标准中“Annex G”提到的问题，这些资源的类型给出了这些问题性质的额外信息（见 5.5.2 节）。

下面给出连接这些资源的关键字及相关解释。

5.5.1 原始参考关键字

Reference	**Source**
Annex G of ISO 9899 [2]	
Unspecified	Unspecified behavivour (G.1)
Undefined	Undefined behavivour (G.2)
Implementation	Implementation-defined behaviour (G.3)
Locale	Locale-specific behaviour (G.4)

Other	
MISRA Guidelines	The MISRA Guidelines [9]
K & R	Kernighan and Ritchie [18]
Koenig	“C Traps and Pitfalls”，Koenig [19]
IEC 61508	IEC 61508: 1998-2002 [20]

参考后面出现数字时，它们具有如下含义：

- ISO 9899 参考的 Annex G：该条款在 Annex 中相应章节中的序号，序号从该章节的开始处计算。所以，[Locale 2]代表标准中 G.4 节的第二条款。
- 在排列参考时给出相应的页数（除非另外声明）。

5.5.2 理解 Annex G 参考

如果规则是基于 ISO C 标准中的 Annex G，理解以下问题的区别将对读者有所帮助，即“unspecified”、“undefined”、“implementation-defined”和“locale-specific”。这里给出简单的解释，详细信息请参考 Hatton[3]。

5.5.2.1 Unspecified

这些是必须成功编译的语言结构，但关于结构的行为，编译器的编写者有某些自由。例如规则 12.2 中描述的“运算次序”。这样的问题有 22 个。

在某种方式上完全相信编译器的行为是不明智的。编译器的行为甚至不会在所有可能的结构中都是一致的。

5.5.2.2 Undefined

这些是本质的编程错误，但编译器的编写者不一定为此给出错误信息。相应的例子是无效参数传递给函数，或函数的参数与定义时的参数不匹配。

从安全性角度这是特别重要的问题，因为它们代表了那些不一定能被编译器捕捉到的错误。

5.5.2.3 Implementation-defined

这有些类似于“unspecified”问题，其主要区别在于编译器要提供一致的行为并记录成文档。换句话说，不同的编译器之间功能可能会有不同，使得代码不具有可移植性，但在任一编译器内，行为应当是良好定义的。比如用在一个正整数和一个负整数上的整除运算“/”和求模运算符“%”。存在 76 个这样的问题。

从安全性角度，假如编译器完全地记录了它的方法并坚持它的实现，那么它可能不是那样至关重要。尽可能的情况下要避免这些问题。

5.5.2.4 Locale-specfic

这是与国际标准有所不同的一些小的特性。如在表示十进制的小数点时使用“,”字符替代“.”字符。有 6 个这样的问题。本文档没有涉及这项资源提出的问题。

5.6 规则的范围

原则上，规则要用在应用程序的所有程序文件。虽然多数规则可以用在单独的文件中，规则 1.1、1.2、1.3、1.4、2.1、3.6、5.1、5.3、5.4、5.5、5.6、5.7、8.4、8.8、8.9、8.10、12.2、12.4、14.1、14.2、16.2、16.4、17.2、18.1、18.2、18.3 和 21.1 必须要用在所有文件中。

6 规则

6.1 环境

规则 1.1（强制）：　所有代码都必须遵照 ISO 9899:1990 “Programming languages - C”，由 ISO/IEC 9899/COR1:1995，ISO/IEC 9899/AMD1:1995，和 ISO/IEC 9899/COR2:1996 修订。

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

这些方针建立在 ISO 9899:1990 [2] 之上，它是由 ISO/IEC 9899/COR1:1995 [4]，ISO/IEC 9899/AMD1:1995 [5] 以及 ISO/IEC 9899/COR2:1996 [6] 修订的。并没有适应于 ISO 9899:1999 [8] 标准版本的声明，本文档中任何提及“Standard C”的地方都是指旧有的 ISO9899:1990 [2] 标准。

必须认识到“背离”（描述见 4.3.2 节）的提出是必要的，这能允许一定的语言扩展，比如对硬件特性的支持。

当 ISO 9899:1990 5.2.4 [2] 中指定的环境限制超出了范围，就需要提出“背离”，除了后面的规则 5.1。

规则 1.2（强制）：　不能有对未定义行为或未指定行为的依赖性。

这项规则要求任何对未定义行为或未指定行为的依赖，除非在其他规则中做了特殊说明，都应该避免。如果其他某项规则中声明了某个特殊行为，那么就只有这项特定规则在其需要时给出背离性。这些问题的完整列表详见 ISO 9899:1990 附录 G [2]。

规则 1.3（强制）：　多个编译器和/或语言只能在为语言/编译器/汇编器所适合的目标代码定义了通用接口标准时使用。

[未指定 11]

如果一个模块是以非 C 语言实现的或是以不同的 C 编译器编译的，那么必须要保证该模块能够正确地同其他模块集成。C 语言行为的某些特征依赖于编译器，于是这些行为必须能够为使用的编译器所理解。例如：栈的使用、参数的传递和数据值的存储方式（长度、排列、别名、覆盖，等等）。

规则 1.4（强制）：　编译器/链接器要确保 31 个有效字符和大小写敏感能被外部标识符支持。

[未定义 7；实现 5、6]

ISO 标准要求外部标识符的头 6 个字符是截然不同的。然而由于大多数编译器/链接器允许至少 31 个有效字符（如同内部标识符），因此对这样严格而并不具有帮助性的限制的适应性被认为是不必要的 。

必须检查编译器/链接器具有这种特性，如果编译器/链接器不能满足这种限制，就使用编译器本身的约束。

规则 1.5（建议）：　浮点应用应该适应于已定义的浮点标准

浮点运算会带来许多问题，一些问题（而不是全部）可以通过适应已定义的标准来克服。其中一个合适的标准是 ANSI/IEEE Std 754 [21]。

同规则 6.3 相一致，浮点类型的定义提供了一个注释所用浮点标准的机会，如：

/* IEEE 754 single-precision floating-point */

typedef float float32_t;

6.2 语言扩展

规则 2.1（强制）： 汇编语言应该被封装并隔离。

[未指定 11]

在需要使用汇编指令的地方，建议以如下方式封装并隔离这些指令：(a) 汇编函数、(b) C 函数、(c) 宏。

出于效率的考虑，有时必须要嵌入一些简单的汇编指令，如开关中断。如果不管出于什么原因需要这样做，那么最好使用宏来完成。

需要注意的是，内嵌的汇编语言的使用是对标准 C 的扩展，因此也需要提出对规则 1.1 的背离。

#define NOP asm (“ NOP”);

规则 2.2（强制）： 源代码应该使用 /*…*/ 类型的注释。

这排除了如 // 这样 C99 类型的注释和 C++类型的注释，因为它在 C90 中是不允许的。许多编译器支持 // 类型的注释以做为对 C90 的扩展。预处理指令（如#define）中 // 的使用可以改变，/*…*/和//的混合使用也是不一致的。这不仅是类型问题，因为不同的编译器（在 C99 之前）可能会有不同的行为。

规则 2.3（强制）： 字符序列 /* 不应出现在注释中。

C 不支持注释的嵌套，尽管一些编译器支持它以做为语言扩展。一段注释以/*开头，直到第一个*/为止，在这当中出现的任何/*都违反了本规则。考虑如下代码段：

/* some comment, end comment marker accidentally omitted

<<New Page>>

Perform_Critical_Safety_Function (X);

/* this comment is not compliant */

在检查包含函数调用的页中，假设它是可执行代码。

因为可能会省略掉注释的结束标记，那么对安全关键函数的调用将不会被执行。

规则 2.4（建议）： 代码段不应被“注释掉”（comment out）。

当源代码段不需要被编译时，应该使用条件编译来完成（如带有注释的#if 或#ifdef 结构）。为这种目的使用注释的开始和结束标记是危险的，因为 C 不支持嵌套的注释，而且已经存在于代码段中的任何注释将影响执行的结果。

6.3 文档

规则 3.1（强制）： 所有实现定义（implementation-defined）的行为的使用都应该文档化。

本规则要求，任何对实现定义的行为的依赖——这些行为在其他规则中没有特别说明的——都应该写成文档，例如对编译器文档的参考。如果一个特定的行为在其他规则中被显式说明了，那么只有那项规则在其需要时给出背离。完整问题的描述详见 ISO 9899:1990 附录 G [2]。

规则 3.2（强制）： 字符集和相应的编码应该文档化。

例如，ISO 10646 [22]定义了字符集映射到数字值的国际标准。出于可移植性的考虑，字符常量和字符串只能包含映射到已经文档化的子集中的字符。

规则 3.3（建议）：　　应该确定、文档化和重视所选编译器中整数除法的实现。

[实现 18]

当两个有符号整型数做除法时，ISO 兼容的编译器的运算可能会为正或为负。首先，它可能以负余数向上四舍五入（如，-5/3 = -1，余数为-2），或者可能以正余数向下四舍五入（如，-5/3 = -2，余数为+1）。重要的是要确定这两种运算中编译器实现的是哪一种，并以文档方式提供给编程人员，特别是第二种情况（通常这种情况比较少）。

规则 3.4（强制）：　　所有#pragma 指令的使用应该文档化并给出良好解释。

[实现 40]

这项规则为本文档的使用者提供了产生其应用中使用的任何 pragma 的要求。每个 pragma 的含义要写成文档，文档中应当包含完全可理解的对 pragma 行为及其在应用中之含义的充分描述。

应当尽量减少任何 pragma 的使用，尽可能地把它们本地化和封装成专门的函数。

规则 3.5（强制）：　　如果做为其他特性的支撑，实现定义（implementation-defined）的行为和位域（bitfields）集合应当文档化。

[未指定 10；实现 30、31]

这是在使用了规则 6.4 和规则 6.5 中描述的非良好定义的位域时遇到的特定问题。C 当中的位域是该语言中最缺乏良好定义的部分之一。位域的使用可能体现在两个主要方面：

- 为了在大的数据类型（同 union 一起）中访问独立的数据位或成组数据位。该用法是不允许的（见规则 18.4）。
- 为了访问用于节省存储空间而打包的标志（flags）或其他短型（short-length）数据。

为了有效利用存储空间而做的短型数据的打包，是本文档所设想的唯一可接受的位域使用。假定结构元素只能以其名字来访问，那么程序员就无需设想结构体中位域的存储方式。

我们建议结构的声明要保持位域的设置，并且在同一个结构中不得包含其他任何数据。要注意的是，在定义位域的时候不需要追随规则 6.3，因为它们的长度已经定义在结构中了。

如果编译器带有一个开关以强制位域遵循某个特定的布局，那么它有助于下面的判断。

例如下面可接受的代码：

```
struct message                          /* Struct is for bit-fields only */
{
    signed   int  little:    4;   /* Note: use of basic types is required */
    unsigned int  x_set:     1;
    unsigned int  y_set:     1;
}message_chunk;
```

如果要使用位域，就得注意实现定义的行为所存在的领域及其潜藏的缺陷（意即不可移植性）。特别地，程序员应当注意如下问题：

- 位域在存储单元中的分配是实现定义（implementation-defined）的，也就是说，它们在存储单元（通常是一个字节）中是高端在后（high end）还是低端在后（low end）的。
- 位域是否重叠了存储单元的界限同样是实现定义的行为（例如，如果顺序存储一个 6 位的域和一个 4 位的域，那么 4 位的域是全部从新的字节开始，还是其中 2 位占据一个字节中的剩余 2 位而其他 2 位开始于下个字节）。

规则 3.6（强制）：　　产品代码中使用的所有库都要适应本文档给出的要求，并且要经过适当的验证。

[IEC 61508 Part 3]

本规则的对象是产品代码中的任意库，因此这些库可能包含编译器提供的标准库、其他第三方的库或者实验室中自己开发的库。这是由 IEC 61508 Part 3 建议的。

6.4 字符集

规则 4.1（强制）：　　只能使用 ISO C 标准中定义的 escape 序列。

[未定义 11；实现 11]

参见 5.2.2 节中关于有效 escape 序列的 ISO 标准。

规则 4.2（强制）：　　不能使用三字母词（trigraphs）。

三字母词由 2 个问号序列后跟 1 个确定字符组成（如，??- 代表“~”（非）符号，而??）代表“]”符号）。它们可能会对 2 个问号标记的其他使用造成意外的混淆，例如字符串

“(Date should be in the form ??-??-??)”

将不会表现为预期的那样，实际上它被编译器解释为

“(Date should be in the form ~~)”

6.5 标识符

规则 5.1（强制）：　　标识符（内部的和外部的）的有效字符不能多于 31。

[未定义 7；实现 5、6]

ISO 标准要求在内部标识符之间前 31 个字符必须是不同的以保证可移植性。即使编译器支持，也不能超出这个限制。

ISO 标准要求外部标识符之间前 6 个字符必须是不同的（忽略大小写）以保证最佳的可移植性。然而这条限制相当严格并被认为不是必须的。本规则的意图是为了在一定程度上放宽 ISO 标准的要求以适应当今的环境，但应当确保 31 个字符/大小写的有效性是可以由实现所支持的。

使用标识符名称要注意的一个相关问题是发生在名称之间只有一个字符或少数字符不同的情况，特别是名称比较长时，当名称间的区别很容易被误读时问题就比较显著，比如 1（数字 1）和 l（L 的小写）、0 和 O、2 和 Z、5 和 S，或者 n 和 h。建议名称间的区别要显而易见。在这问题上的特定方针可以放在风格指南中（见 4.2.2 节）。

规则 5.2（强制）：　　具有内部作用域的标识符不应使用与具有外部作用域的标识符相同的名称，这会隐藏了外部标识符。

外部作用域和内部作用域的定义如下。文件范围内的标识符可以看做是具有最外部（outermost）的作用域；块范围内的标识符看做是具有更内部（more inner）的作用域；连续嵌套的块，其作用域更深入。本规则只是不允许一个第二深层（second inner）的定义隐藏其外层的定义，如果第二个定义没有隐藏第一个定义，那么就不算违反本规则。

在嵌套的范围中，使用相同名称的标识符隐藏其他标识符会使得代码非常混乱。例如：

```
int16_t    i;
{
    int16_t    i;    /* This is a different variable                    */
                     /* This is not compliant                           */
    i = 3;           /* It could be confusing as to which I this refers  */
}
```

规则 5.3（强制）：　　typedef 的名字应当是唯一的标识符。

typedef 的名称不能重用，不管是做为其他 typedef 或者任何目的。例如：

```
{
    typedef    unsigned char uint8_t;
}
{
    typedef    unsigned char uint8_t;   /* Not compliant – redefinition */
}
{
    unsigned char uint8_t;              /* Not compliant – reuse of uint8_t */
}
```

typedef 的名称不能在程序中的任何地方重用。如果类型定义是在头文件中完成的，而该头文件被多个源文件包含，不算违背本规则。

规则 5.4（强制）：　　标签（tag）名称必须是唯一的标识符。

程序中标签的名字不可重用，不管是做为另外的标签还是出于其他目的。ISO 9899:1990 [2] 没有定义当一个聚合体的声明以不同形式的类型标识符（struct 或 union）使用同一个标签时的行为。标签的所有使用或者用于结构类型标识符，或者用于联合类型标识符，例如：

```
struct    stag    { uint16_t    a;    uint16_t    b; };
struct    stag    a1 = { 0, 0 };  /* Compliant – compatible with above      */
union     stag    a2 = { 0, 0 };  /* Not compliant – not compatible with
                                     previous declarations */
void foo (void)
{
    struct    stag    { uint16_t    a; };    /* Not compliant – tag stag redefined */
}
```

如果类型定义是在头文件中完成的，且头文件被多个源文件包含，那么规则不算违背。

规则 5.5（建议）：　　具有静态存储期的对象或函数标识符不能重用。

不管作用域如何，具有静态存储期的标识符都不应在系统内的所有源文件中重用。它包含带有外部链接的对象或函数，及带有静态存储类标识符的任何对象或函数。

由于编译器能够理解这一点而且决不会发生混淆，那么对用户来说就存在着把不相关的变量以相同名字联系起来的可能性。

这种混淆的例子之一是，在一个文件中存在一个具有内部链接的标识符，而在另外一个文件中存在着具有外部链接的相同名字的标识符。

规则 5.6（建议）：　　一个命名空间中不应存在与另外一个命名空间中的标识符拼写相同的标识符，除了结构和联合中的成员名字。

命名空间与作用域（scope）是不同的，本规则不考虑作用域。例如，ISO C 允许在一个作用域内为标签（tag）和 typedef 使用相同的标识符（vector）

```
typedef    struct    vector ( uint16_t    x ;    uint16_t   y ;    uint16_t   z ; ) vector ;
/* Rule violation        ^^                                                ^^      */
```

ISO C 定义了许多不同的命名空间（见 ISO 9899 :1990 6.1.2.3 [2]）。技术上，在彼此独立的命名空间中使用相同的名字以代表完全不同的项目是可能的，然而由于会引起混淆，通常

不赞成这种做法，因此即使是在独立的命名空间中名字也不能重用。

下面给出了违背此规则的例子，其中 value 在不经意中代替了 record.value：

```
struct      { int16_t  key ;       int16_t     value ; } record ;
int16_t     value;      /* Rule violation – 2nd use of value */
record.key = 1;
value = 0;        /* should have been record.value */
```

相比之下，下面的例子没有违背此规则，因为两个成员名字不会引起混淆：

```
struct      device_q { struct     device_q *next ;     /* ... */ }
devices[N_DEVICES] ;
struct      task_q { struct       task_q *next ;       /* … */ }
tasks[N_TASKS];
device[0].next = &devices[1];
tasks[0].next = &tasks[1];
```

规则 5.7（建议）：　　不能重用标识符名字。

不考虑作用域，系统内任何文件中不应重用标识符。本规则和规则 5.2、5.3、5.4、5.5 和 5.6 一同使用。

```
struct      air_speed
{
      uint16_t   speed;     /* knots */
} *x;
struct      gnd_speed
{
      uint16_t   speed;     /* mph                                          */
                            /* Not Compliant – speed is in different units */
} *y;
x->speed = y->speed;
```

当标识符名字用在头文件且头文件包含在多个源文件中时，不算违背本规则。使用严格的命名规范可以支持本规则。

6.6　类型

规则 6.1（强制）：　　单纯的 char 类型应该只用做存储和使用字符值。

[实现　14]

规则 6.2（强制）：　　signed char 和 unsigned char 类型应该只用做存储和使用数字值。

有三种不同的 char 类型：（单纯的）char、unsigned char、signed char。unsigned char 和 signed char 用于数字型数据，char 用于字符型数据。单纯 char 类型的符号是实现定义的，不应依赖。

单纯 char 类型所能接受的操作只有赋值和等于操作符（=、==、!=）。

规则 6.3（建议）：　　应该使用指示了大小和符号的 typedef 以代替基本数据类型。

不应使用基本数值类型 char、int、short、long、float 和 doulbe，而应使用特定长度（specific-length）的 typedef。规则 6.3 帮助我们认清存储类型的大小，却不能保证可移植性，

这是因为整数提升（integral promotion）的不对称性。关于整数提升的讨论，见节 6.10。仍然很重要的是要理解整数大小的实现。

程序员应该注意这些定义之下的 typedef 的实际实现。

比如，本文档中建议为所有基本数值类型和字符类型使用如下所示的 ISO（POSIX）的 typedef。对于 32 位计算机，它们是：

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

在位域类型的说明中，typedef 是不必要的。

规则 6.4（强制）：　　位域只能被定义为 unsigned int 或 singed int 类型。

[未定义 38；实现 29]

因为 int 类型的位域可以是 signed 或 unsigned，使用 int 是由实现定义的。由于其行为未被定义，所以不允许为位域使用 enum、short 或 char 类型。

规则 6.5（强制）：　　unsigned int 类型的位域至少应该为 2 bits 长度。

1 bit 长度的有符号位域是无用的。

6.7　常量

规则 7.1（强制）：　　不应使用八进制常量（零除外）和八进制 escape 序列。

[Koenig 9]

任何以“0”（零）开始的整型常量都被看做是八进制的，所以这是危险的，如在书写固定长度的常量时。例如，下面为 3 个数字位的总线消息做数组初始化时将产生非预期的结果（052 是八进制的，即十进制的 42）：

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

八进制的 escape 序列是有问题的，这是因为在八进制 escape 结尾不经意引入一个十进制数会产生另外一个字符。下面例子中，第一个表达式的值是实现定义的，因为其字符常量包含了两个字符，“\10”和“9”。第二个字符常量表达式包含了单一字符“\100”，如果字符 64 不在基本运算字符集中，这也将是由实现定义的。

```
code[5] = '\109' ;   /* implementation-defined, two character constant */
```

```
code[6] = '\100' ;   /* set to 64, or implementation-defined          */
```

最好根本不要使用八进制常量或 escape 序列，并且要静态检查它们是否出现。整数常量 0（做为单个数字书写的）严格说来是八进制常量，然而在此规则下它也是允许的。

6.8 声明与定义

规则 8.1（强制）：　函数应当具有原型声明，且原型在函数的定义和调用范围内都是可见的。

[未定义 22、23]

原型的使用使得编译器能够检查函数定义和调用的完整性。如果没有原型，就不会迫使编译器检查出函数调用当中的一定错误（比如，函数体具有不同的参数数目，调用和定义之间参数类型的不匹配）。事实证明，函数接口是相当多问题的肇因，因此本规则是相当重要的。

对外部函数来说，我们建议采用如下方法，在头文件中声明函数（亦即给出其原型），并在所有需要该函数原型的代码文件中包含这个头文件（见规则 8.8）。

为具有内部链接的函数给出其原型也是良好的编程实践。

规则 8.2（强制）：　不论何时声明或定义了一个对象或函数，它的类型都应显式声明。

```
extern          x;          /* Non-compliant – implicit int type    */
extern  int16_t x ;         /* Compliant – explicit type            */
const           y ;         /* Non-compliant – implicit int type    */
const   int16_t y ;         /* Compliant – explicit type            */
static          foo (void) ; /* Non-compliant – implicit type       */
static  int16_t foo (void) ; /* Compliant – explicit type           */
```

规则 8.3（强制）：　函数的每个参数类型在声明和定义中必须是等同的，函数的返回类型也该是等同的。

[未定义 24；Koenig 59 – 62]

参数与返回值的类型在原型和定义中必须匹配，这不仅要求等同的基本类型，也要求包含 typedef 名称和限定词在内的类型也要相同。

规则 8.4（强制）：　如果对象或函数被声明了多次，那么它们的类型应该是兼容的。

[未定义 10]

兼容类型的定义是冗长复杂的（详见 ISO 9899 :1990 [2]，节 6.1.2.6、6.5.2、6.5.3、6.5.4）。两个等同的类型必然是兼容的，而两个兼容的类型不需要等同。例如，下面的类型对是兼容的：

```
signed   int              int
char[5]                   char []
unsigned short   int      unsigend short
```

规则 8.5（强制）：　头文件中不应有对象或函数的定义。

头文件应该用于声明对象、函数、typedef 和宏，而不应该包含或生成占据存储空间的对象或函数（或它们的片断）的定义。这样就清晰地划分了只有 C 文件才包含可执行的源代码，而头文件只能包含声明。

规则 8.6（强制）：　函数应该声明为具有文件作用域。

[未定义 36]

在块作用域中声明函数会引起混淆并可能导致未定义的行为。

规则 8.7（强制）：　　如果对象的访问只是在单一的函数中，那么对象应该在块范围内声明。

可能的情况下，对象的作用域应该限制在函数内。只有当对象需要具有内部或外部链接时才能为其使用文件作用域。当在文件范围内声明对象时，使用规则 8.10。良好的编程实践是，在不必要的情况下避免使用全局标识符。对象声明在最外层或最内层的做法主要是种风格问题。

规则 8.8（强制）：　　外部对象或函数应该声明在唯一的文件中。

[Koenig 66]

通常这意味着在一个头文件中声明一个外部标识符，而在定义或使用该标识符的任何文件中包含这个头文件。例如，在头文件 featureX.h 中声明：

```
extern     int16_t    a ;
```

然后对 a 进行定义：

```
#include  <featureX.h>

int16_t    a = 0 ;
```

工程中存在的头文件可能是一个或多个，但是任何一个外部对象或函数都只能在一个头文件中声明。

规则 8.9（强制）：　　具有外部链接的标识符应该具有准确的外部定义。

[未定义 44；Koenig 55、63 – 65]

一个标识符如果存在多个定义（在不同的文件中）或者甚至没有定义，那么其行为是未经定义的。不同文件中的多个定义是不允许的，即使这些定义相同也不允许；进而如果这些定义不同或者标识符的初始值不同，问题显然很严重。

规则 8.10（强制）：　　在文件范围内声明和定义的所有对象或函数应该具有内部链接，除非是在需要外部链接的情况下。

[Koenig 56、57]

如果一个变量只是被同一文件中的函数所使用，那么就用 static。类似地，如果一个函数只是在同一文件中的其他地方调用，那么就用 static。使用 static 存储类标识符将确保标识符只是在声明它的文件中是可见的，并且避免了和其他文件或库中的相同标识符发生混淆的可能性。

规则 8.11（强制）：　　static 存储类标识符应该用于具有内部链接的对象和函数的定义和声明。

static 和 extern 存储类标识符常常是产生混淆的原因。良好的编程习惯是，把 static 关键字一致地应用在所有具有内部链接的对象和函数的声明上。

规则 8.12（强制）：　　当一个数组声明为具有外部链接，它的大小应该显式声明或者通过初始化进行隐式定义。

[未定义 46]

```
int    array1[10] ;                  /* Compliant          */
extern     int    array2[] ;         /* Not compliant      */
int    array2[] = { 0, 10, 15 };     /* Compliant          */
```

尽管可以在数组声明不完善时访问其元素，然而仍然是在数组的大小可以显式确定的情况下，这样做才会更为安全。

6.9 初始化

规则 9.1（强制）：　　所有自动变量在使用前都应被赋值。

[未定义 41]

本规则的意图是使所有变量在其被读之前已经写过了，除了声明中的初始化。

注意，根据 ISO C 标准，具有静态存储期的变量缺省地被自动赋予零值，除非经过了显式的初始化。实际中，一些嵌入式环境没有实现这样的缺省行为。静态存储期是所有以 static 存储类形式声明的变量或具有外部链接的变量的共同属性，自动存储期变量通常不是自动初始化的。

规则 9.2（强制）：　　应该使用大括号以指示和匹配数组和结构的非零初始化构造。

[未定义 42]

ISO C 要求数组、结构和联合的初始化列表要以一对大括号括起来（尽管不这样做的行为是未定义的）。本规则更进一步地要求，使用附加的大括号来指示嵌套的结构。它迫使程序员显式地考虑和描述复杂数据类型元素（比如，多维数组）的初始化次序。

例如，下面的例子是二维数组初始化的有效（在 ISO C 中）形式，但第一个与本规则相违背：

```
int16_t    y[3][2] = { 1, 2, 3, 4, 5, 6 };                  /* not compliant    */
int16_t    y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } } ;     /* compliant        */
```

在结构中以及在结构、数组和其他类型的嵌套组合中，规则类似。

还要注意的是，数组或结构的元素可以通过只初始化其首元素的方式初始化（为 0 或 NULL）。如果选择了这样的初始化方法，那么首元素应该被初始化为 0（或 NULL），此时不需要使用嵌套的大括号。

ISO 标准 [2] 包含了更多的初始化例子。

规则 9.3（强制）：　　在枚举列表中，“=”不能显式用于除首元素之外的元素上，除非所有的元素都是显式初始化的。

如果枚举列表的成员没有显式地初始化，那么 C 将为其分配一个从 0 开始的整数序列，首元素为 0，后续元素依次加 1。

如上规则允许的，首元素的显式初始化迫使整数的分配从这个给定的值开始。当采用这种方法时，重要的是确保所用初始化值一定要足够小，这样列表中的后续值就不会超出该枚举常量所用的 int 存储量。

列表中所有项目的显式初始化也是允许的，它防止了易产生错误的自动与手动分配的混合。然而，程序员就该担负职责以保证所有值都处在要求的范围内以及值不是被无意复制的。

```
enum      colour { red = 3, blue, green, yellow = 5 } ;  /* not compliant */
/* green and yellow represent the same value – this is duplication          */
enum      colour { red = 3, blue = 4, green = 5, yellow = 5 };      /* compliant */
/* green and yellow represent the same value – this is duplication          */
```

6.10 数值类型转换

6.10.1 隐式和显式类型转换

C 语言给程序员提供了相当大的自由度并允许不同数值类型可以自动转换。由于某些功能性的原因可以引入显式的强制转换，例如：

- 用以改变类型使得后续的数值操作可以进行
- 用以截取数值
- 出于清晰的角度，用以执行显式的类型转换

为了代码清晰的目的而插入的强制转换通常是有用的，但如果过多使用就会导致程序的可读性下降。正如下面所描述的，一些隐式转换是可以安全地忽略的，而另一些则不能。

6.10.2 隐式转换的类型

存在三种隐式转换的类别需要加以区分。

整数提升（Integral promotion）转换

整数提升描述了一个过程，借此过程数值操作总是在 int 或 long（signed 或 unsigned）整型操作数上进行。其他整型操作数（char、short、bit-field 和 enum）在数值操作前总是先转化为 int 或 unsigned int 类型。这些类型称为 small integer 类型。

整数提升的规则命令，在大多数数值操作中，如果 int 类型能够代表原来类型的所有值，那么 small integer 类型的操作数要被转化为 int 类型；否则就被转化为 unsigned int。

注意，整数提升：

- 仅仅应用在 small integer 类型上
- 应用在一元、二元和三元操作数上
- 不能用在逻辑操作符（&&、||、!）的操作数上
- 用在 switch 语句的控制表达式上

整数提升经常和操作数的“平衡”（balancing，后面提到）发生混淆。事实上，整数提升发生在一元操作的过程中，如果二元操作的两个操作数是同样类型的，那么也可以发生在二元操作之上。

由于整数提升，两个类型为 unsigned short 的对象相加的结果总是 signed int 或 unsigned int 类型的；事实上，加法是在后面两种类型上执行的。因此对于这样的操作，就有可能获得一个其值超出了原始操作数类型大小的结果。例如，如果 int 类型的大小是 32 位，那么就能够把两个 short（16 位）类型的对象相乘并获得一个 32 位的结果，而没有溢出的危险。另一方面，如果 int 类型仅是 16 位，那么两个 16 位对象的乘积将只能产生一个 16 位的结果，同时必须对操作数的大小给出适当的限制。

整数提升还可以应用在一元操作符上。例如，对一个 unsigned char 操作数执行位非（~）运算，其结果典型地是 signed int 类型的负值。

整数提升是 C 语言中本质上的不一致性，在这当中 small integer 类型的行为与 long 和 int 类型不同。MISRA-C 鼓励使用 typedef。然而，由于众多整型的行为是不一致的，忽略基本类型（见后面的描述）可能是不安全的，除非对表达式的构造方式给出一些限制。后面规则的意图是想中和整数提升的后果以避免这些异常。

赋值转换

赋值转换发生在：

- 赋值表达式的类型被转化成赋值对象的类型时
- 初始化表达式的类型被转化成初始化对象的类型时
- 函数调用参数的类型被转化成函数原型中声明的形式参数的类型时
- 返回语句中用到的表达式的类型被转化成函数原型中声明的函数类型时
- switch-case 标签中的常量表达式的类型被转化成控制表达式的提升类型时。这个转换仅用于比较的目的

每种情况中，必要时数值表达式的值是无条件转换到其他类型的。

平衡转换（Balancing conversions）

平衡转换的描述是在 ISO C 标准中的“Usual Arithmetic Conversions”条目下。这套规则提供一个机制，当二元操作符的两个操作数要平衡为一个通用类型时或三元操作符（?:）的第二、第三个操作数要平衡为一个通用类型时，产生一个通用类型。平衡转换总是涉及到两个不同类型的操作数；其中一个、有时是两个需要进行隐式转换。整数提升（上面描述的）的过程使得平衡转换规则变得复杂起来，在整数提升时，small integer 类型的操作数首先要提升到 int 或 unsigned int 类型。整数提升是常见的数值转换，即使两个操作数的类型一致。

与平衡转换明显相关的操作符是：

- 乘除 *、/、%
- 加减 +、-
- 位操作 &、^、|
- 条件操作符 （…？…：…）
- 关系操作符 >、>=、<、<=
- 等值操作符 ==、!=

其中大部分操作符产生的结果类型是由平衡过程产生的，除了关系和等值操作符，它们产生具有 int 类型的布尔值。

要注意的是，位移操作符（<<和>>）的操作数不进行平衡，运算结果被提升为第一个操作数的类型；第二个操作数可以是任何有符号或无符号的整型。

6.10.3 危险的类型转换

类型转换过程中存在大量潜在的危险需要加以避免:

- 数值的丢失：转化后的类型其数值量级不能被体现
- 符号的丢失：从有符号类型转换为无符号类型会导致符号的丢失
- 精度的丢失：从浮点类型转换为整型会导致精度的丢失

对于所有数据和所有可能的兼容性实现来说，唯一可以确保为安全的类型转换是：

- 整数值进行带符号的转换到更宽类型
- 浮点类型转换到更宽的浮点类型

当然，在实践中，如果假定了典型类型的大小，也能够把其他类型转换归类为安全的。普遍来说，MISRA-C:2004 采取的原则是，利用显式的转换来辨识潜藏的危险类型转换。

类型转换中还有其他的一些危险需要认清。这些问题产生于 C 语言的难度和误解，而不是由于数据值不能保留。

- 整数提升中的类型放宽：整数表达式运算的类型依赖于经过整数提升后的操作数的类型。总是能够把两个 8 位数据相乘并在有量级需要时访问 16 位的结果。有时而不总是能够把两个 16 位数相乘并得到一个 32 位结果。这是 C 语言中比较危险的不一致性，为了避免混淆，安全的做法是不要依赖由整数提升所提供的类型放宽。考虑如下例子：

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

期望的结果是 70000，但是赋给 u 的值在实际中依赖于 int 实现的大小。如果 int 实现的大

小是 32 位，那么加法就会在有符号的 32 位数值上运算并且保存下正确的值。如果 int 实现的大小仅是 16 位，那么加法会在无符号的 16 位数值上进行，于是会发生折叠（wraparound）现象并产生值 4464（70000%65536）。无符号数值的折叠（wraparound）是经过良好定义的甚至是有意的；但也会存在潜藏的混淆。

- 类型计算的混淆：程序员中常见的概念混乱也会产生类似的问题，人们经常会以为参与运算的类型在某种方式上受到被赋值或转换的结果类型的影响。例如，在下面的代码中，两个 16 位对象进行 16 位的加法运算（除非被提升为 32 位 int），其结果在赋值时被转换为 uint32_t 类型。

  u32x = u16a + u16b;

  并非少见的是，程序员会认为此表达式执行的是 32 位加法——因为 u32x 的类型。

  对这种特性的混淆不只局限于整数运算或隐式转换，下面的例子描述了在某些语句中，结果是良好定义的但运算并不会按照程序员设想的那样进行。

  u32a = (uint32_t) (u16a * u16b);

  f64a = u16a / u16b ;

  f32a = (float32_t) (u16a / u16b) ;

  f64a = f32a + f32b ;

  f64a = (float64_t) (f32a + f32b) ;

- 数学运算中符号的改变：整数提升经常会导致两个无符号的操作数产生一个（signed）int 类型的结果。比如，如果 int 是 32 位的，那么两个 16 位无符号数的加法将产生一个有符号的 32 位结果；而如果 int 是 16 位的，那么同样运算会产生一个无符号的 16 位结果。

- 位运算中符号的改变：当位运算符应用在无符号短整型时，整数提升会有某些特别不利的反响。比如，在一个 unsigned char 类型的操作数上做位补运算通常会产生其值为负的（signed）int 类型结果。在运算之前，操作数被提升为 int 类型，并且多出来的那些高位被补运算置 1。那些多余位的个数，若有的话，依赖于 int 的大小，而且在补运算后接右移运算是危险的。

为了避免上述问题产生的危险，重要的是要建立一些准则以限制构建表达式的方式。这里首先给出某些概念的定义。

6.10.4 基本类型（underlying type）

表达式的类型是指其运算结果的类型。当两个 long 类型的值相加时，表达式具有 long 类型。大多数数值运算符产生其类型依赖于操作数类型的结果。另一方面，某些操作符会给出具有 int 类型的布尔结果而不管其操作数类型如何。所以，举例来说，当两个 long 类型的项用关系运算符做比较时，该表达式的类型为 int。

术语“基本类型”的定义是，在不考虑整数提升的作用下描述由计算表达式而得到的类型。

当两个 int 类型的操作数相加时，结果是 int 类型，那么表达式可以说具有 int 类型。

当两个 unsigned char 类型的数相加时，结果也是 int 类型（通常如此，因为整数提升的原因），但是该表达式**基本的**类型按照定义则是 unsigned char。

术语“基本类型”不是 C 语言标准或其他 C 语言的文本所认知的，但在描述如下规则时它很有用。它描述了一个假想的对 C 语言的违背，其中不存在整数提升而且常用的数值转换一致性地应用于所有的整数类型。引进这样的概念是因为整数提升很敏感且有时是危险的。整数提升是 C 语言中不可避免的特性，但是**这些规则的意图是要使整数提升的作用能够通过不利用发生在 small integer 操作数上的宽度扩展来中和。**

当然，C 标准没有显式地定义在缺乏整数提升时 small integer 类型如何平衡为通用类型，尽管标准确实建立了值保留（value-perserving）原则。

当 int 类型的数相加时，程序员必须要确保运算结果不会超出 int 类型所能体现的值。如果他没有这样做，就可能会发生溢出而结果值是未定义的。这里描述的方法意欲使得在做 small integer 类型的加法时使用同样的原则；程序员要确保两个 unsigned char 类型的数相加的结果是能够被 unsigned char 体现的，即使整数提升会引起更大类型的计算。换句话说，对表达式基本类型的遵守要多于其真实类型。

整数常量表达式的基本类型

C 语言的一个不利方面是，它不能定义一个 char 或 short 类型的整型常量。比如，值“5”可以通过附加的一个合适的后缀来表示为 int、unsigned int、long 或 unsigned long 类型的常量；但没有合适的后缀用来创建不同的 char 或 short 类型的常量形式。这为维护表达式中的类型一致性提出了困难。如果需要为一个 unsigned char 类型的对象赋值，那么或者要承受对一个整数类型的隐式转换，或者要实行强制转换。很多人会辩称在这种情况下使用强制转换只能导致可读性下降。

在初始化、函数参数或数值表达式中需要常量时也会遇到同样的问题。然而只要遵守强类型（strong typing）原则，这个问题就会比较乐观。

解决该问题的一个方法是，**想象**整型常量、枚举常量、字符常量或者整型常量表达式具有适合其量级的类型。这个目标可以通过以下方法达到，即延伸基本类型的概念到整型常量上，并想象在可能的情况下数值常量已经通过提升想象中的具有较小基本类型的常量而获得。

这样，整型常量表达式的基本类型就可以如下定义：

1. 如果表达式的真实类型是（signed）int，其基本类型就是能够体现其值的最小的有符号整型。
2. 如果表达式的真实类型是 unsigned int，其基本类型就是能够体现其值的最小的无符号整型。
3. 在所有其他情况下，表达式的基本类型与其真实类型相同。

在常规的体系结构中，整型常量表达式的基本类型可以根据其量级和符号确定如下：

无符号值

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

有符号值

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

注意，基本类型是人造的概念，它不会以任何方式影响实际运算的类型。这个概念的提出只是为了定义一个在其中可以构建数值表达式的安全框架。

6.10.5 复杂表达式（complex expressions）

后面章节中描述的类型转换规则在某些地方是针对“复杂表达式”的概念。术语“复杂表达式”意味着任何不是如下类型的表达式：

- 非常量表达式
- lvalue（即一个对象）
- 函数的返回值

应用在复杂表达式的转换也要加以限制以避免上面总结出的危险。特别地，表达式中的数值运算序列需要以相同类型进行。

以下是复杂表达式：

s8a + s8b

~u16a

u16a >> 2

foo (2) + u8a

*ppc + 1

++u8a

以下不是复杂表达式，尽管某些包含了复杂的子表达式：

pc[u8a]

foo (u8a + u8b)

++ppuc

** (ppc + 1)

pcbuf[s16a * 2]

6.10.6 隐式类型转换

规则 10.1（强制）： **下列条件成立时，整型表达式的值不应隐式转换为不同的基本类型：**

a) 转换不是带符号的向更宽整数类型的转换，或者

b) 表达式是复杂表达式，或者

c) 表达式不是常量而是函数参数，或者

d) 表达式不是常量而是返回的表达式。

规则 10.2（强制）： **下列条件成立时，浮点类型表达式的值不应隐式转换为不同的类型：**

a) 转换不是向更宽浮点类型的转换，或者

b) 表达式是复杂表达式，或者

c) 表达式是函数参数，或者

d) 表达式是返回表达式。

还要注意，在描述整型转换时，始终关注的是基本类型而非真实类型。

这两个规则广泛地封装了下列原则：

- 有符号和无符号之间没有隐式转换
- 整型和浮点类型之间没有隐式转换
- 没有从宽类型向窄类型的隐式转换
- 函数参数没有隐式转换
- 函数的返回表达式没有隐式转换
- 复杂表达式没有隐式转换

限制复杂表达式的隐式转换的目的，是为了要求在一个表达式里的数值运算序列中，所有的运算应该准确地以相同的数值类型进行。注意这并不是说表达式中的所有操作数必须具备相同的类型。

表达式 u32a + u16b + u16c 是合适的——两个加法**在概念上（notionally）**都以 U32 类型进行

表达式 u16a + u16b + u32c 是不合适的——第一个加法**在概念上**以 U16 类型进行，第二个

加法是 U32 类型的。

使用名词“在概念上”是因为，在实际中数值运算的类型将依赖于 int 实现的大小。通过遵循这样的原则，所有运算都以一致的（基本）类型来进行，能够避免程序员产生的混淆和与整数提升有关的某些危险。

```
extern     void foo1 (uint8_t   x);
int16_t    t1 (void)
{
    …
    foo1 (u8a);                 /* compliant */
    foo1 (u8a + u8b);           /* compliant */
    foo1 (s8a);                 /* not compliant */
    foo1 (u16a);                /* not compliant */
    foo1 (2);                   /* not compliant */
    foo1 (2U);                  /* compliant */
    foo1 ( (uint8_t) 2 );       /* compliant */
    … s8a + u8a                 /* not compliant */
    … s8a + (int8_t) u8a        /* compliant */
    s8b = u8a;                  /* not compliant */
    … u8a + 5                   /* not compliant */
    … u8a + 5U                  /* compliant */
    … u8a + (uint8_t) 5         /* compliant */
    u8a = u16a;                 /* not compliant */
    u8a = (uint8_t) u16a;       /* compliant */
    u8a = 5UL;                  /* not compliant */
    … u8a + 10UL                /* compliant */
    u8a = 5U;                   /* compliant */
    … u8a + 3                   /* not compliant */
    … u8a >> 3                  /* compliant */
    … u8a >> 3U                 /* compliant */
    pca = “P”;                  /* compliant */
    … s32a + 80000              /* compliant */
    … s32a + 80000L             /* compliant */
    f32a = f64a;                /* not compliant */
    f32a = 2.5;                 /* not compliant –
                                   unsuffixed floating
                                   constants are of type
                                   double */
    u8a = u8b + u8c;            /* compliant */
    s16a = u8b + u8b;           /* not compliant */
    s32a = u8b + u8c;           /* not compliant */
```

```
        f32a = 2.5F;                    /* compliant */
        u8a = f32a;                     /* not compliant */
        s32a = 1.0;                     /* not compliant */
        f32a = 1;                       /* not compliant */
        f32a = s16a;                    /* not compliant */
        … f32a + 1                      /* not compliant */
        … f64a * s32a                   /* not compliant */
        …
        return (s32a);                  /* not compliant */
        …
        return (s16a);                  /* compliant */
        …
        return (20000);                 /* compliant */
        …
        return (20000L);                /* not compliant */
        …
        return (s8a);                   /* not compliant */
        …
        return (u16a);                  /* not compliant */
}
int16_t     foo2 (void)
{
        …
        … (u16a + u16b) + u32a          /* not compliant */
        … s32a + s8a + s8b              /* compliant */
        … s8a + s8b + s32a              /* not compliant */
        f64a = f32a + f32b;             /* not compliant */
        f64a = f64b + f32a;             /* compliant */
        f64a = s32a / s32b;             /* not compliant */
        u32a = u16a + u16a;             /* not compliant */
        s16a = s8a;                     /* compliant */
        s16a = s16b + 20000;            /* compliant */
        s32a = s16a + 20000;            /* not compliant */
        s32a = s16a + (int32_t) 20000;      /* compliant */
        u16a = u16b + u8a;                  /* compliant */
        foo1 (u16a);                        /* not compliant */
        foo1 (u8a + u8b);                   /* compliant */
        …
        return s16a;                        /* compliant */
        …
```

```
    return s8a;                         /* not compliant */
}
```

6.10.7 显式转换（强制转换）

规则 10.3（强制）：　整型复杂表达式的值只能强制转换到更窄的类型且与表达式的基本类型具有相同的符号。

规则 10.4（强制）：　浮点类型复杂表达式的值只能强制转换到更窄的浮点类型。

如果强制转换要用在任何复杂表达式上，可以应用的转换的类型应该严格限制。如 6.10 节所阐释的，复杂表达式的转换经常是混淆的来源，保持谨慎是明智的做法。为了符合这些规则，有必要使用临时变量并引进附加的语句。

```
… (float32_t) (f64a + f64b)             /* compliant */
… (float64_t) (f32a + f32b)             /* not compliant */
… (float64_t) f32a                      /* compliant */
… (float64_t) (s32a / s32b)             /* not compliant */
… (float64_t) (s32a > s32b)             /* not compliant */
… (float64_t) s32a / (float32_t) s32b   /* compliant */
… (uint32_t) (u16a + u16b)              /* not compliant */
… (uint32_t) u16a + u16b                /* compliant */
... (uint32_t) u16a + (uint32_t) u16b   /* compliant */
... (int16_t) (s32a – 12345)            /* compliant */
... (uint8_t) (u16a * u16b)             /* compliant */
... (uint16_t) (u8a * u8b)              /* not compliant */
... (int16_t) (s32a * s32b)             /* compliant */
... (int32_t) (s16a * s16b)             /* not compliant */
… (uint16_t) (f64a + f64b)              /* not compliant */
… (float32_t) (u16a + u16b)             /* not compliant */
… (float64_t) foo1 (u16a + u16b)        /* compliant */
… (int32_t) buf16a[u16a + u16b]         /* compliant */
```

规则 10.5（强制）：　如果位运算符 ~ 和 << 应用在基本类型为 unsigned char 或 unsigned short 的操作数，结果应该立即强制转换为操作数的基本类型。

当这些操作符（~和<<）用在 small integer 类型（unsigned char 或 unsigned short）时，运算之前要先进行整数提升，结果可能包含并非预期的高端数据位。例如：

```
uint8_t    port = 0x5aU;
uint8_t    result_8;
uint16_t   result_16;
uint16_t   mode;
result_8 = (~port) >> 4;        /* not compliant */
```

~port 的值在 16 位机器上是 0xffa5，而在 32 位机器上是 0xffffffa5。在每种情况下，result 的值是 0xfa，然而期望值可能是 0x0a。这样的危险可以通过如下所示的强制转换来避免：

```
result_8 = ( ( uint8_t ) (~port ) ) >> 4;                   /* compliant */
result_16 = ( ( uint16_t ) (~(uint16_t) port ) ) >> 4 ;     /* compliant */
```

当<<操作符用在 small integer 类型时会遇到类似的问题，高端数据位被保留下来。例如：

```
result_16 = ( ( port << 4 ) & mode ) >> 6 ;                /* not compliant */
```

result_16 的值将依赖于 int 实现的大小。附加的强制转换可以避免任何模糊性。

```
result_16 = ( ( uint16_t ) ( ( uint16_t ) port << 4 ) & mode ) >> 6 ;     /* compliant */
```

6.10.8 整数后缀

规则 10.6（强制）：　后缀“U”应该用在所有 unsigned 类型的常量上。

整型常量的类型是混淆的潜在来源，因为它依赖于许多因素的复杂组合，包括：

- 常数的量级
- 整数类型**实现**的大小
- 任何后缀的存在
- 数值表达的进制（即十进制、八进制或十六进制）

例如，整型常量“40000”在 32 位环境中是 int 类型，而在 16 位环境中则是 long 类型。值 0x8000 在 16 位环境中是 unsigned int 类型，而在 32 位环境中则是（signed）int 类型。

注意：

- 任何带有“U”后缀的值是 unsigned 类型
- 一个不带后缀的小于 2^{31} 的十进制值是 signed 类型

但是：

- 不带后缀的大于或等于 2^{15} 的十六进制数可能是 signed 或 unsigned 类型
- 不带后缀的大于或等于 2^{31} 的十进制数可能是 signed 或 unsigned 类型

常量的符号应该明确。符号的一致性是构建良好形式的表达式的重要原则。如果一个常数是 unsigned 类型，为其加上“U”后缀将有助于避免混淆。当用在较大数值上时，后缀也许是多余的（在某种意义上它不会影响常量的类型）；然而后缀的存在对代码的清晰性是种有价值的帮助。

6.11 指针类型转换

指针类型可以归为如下几类：

- 对象指针
- 函数指针
- void 指针
- 空（null）指针常量（即由数值 0 强制转换为 void*类型）

涉及指针类型的转换需要明确的强制，除非在以下时刻：

- 转换发生在对象指针和 void 指针之间，而且目标类型承载了源类型的所有类型标识符
- 当空指针常量（void*）被赋值给任何类型的指针或与其做等值比较时，空指针常量被自动转化为特定的指针类型

C 当中只定义了一些特定的指针类型转换，而一些转换的行为是实现定义的。

规则 11.1（强制）：　转换不能发生在函数指针和其他除了整型之外的任何类型指针之间。

[未定义 27、28]

函数指针到不同类型指针的转换会导致未定义的行为。举个例子，这意味着一个函数指针不能转换成指向不同类型函数的指针。

规则 11.2（强制）：　对象指针和其他除整型之外的任何类型指针之间、对象指针和其他类型对象的指针之间、对象指针和 void 指针之间不能进行转换。

[未定义 29]

这些转换未经定义。

规则 11.3（建议）：　不应在指针类型和整型之间进行强制转换

[实现 24]

当指针转换到整型时所需要的整型的大小是实现定义的。尽可能的情况下要避免指针和整型之间的转换，但是在访问内存映射寄存器或其他硬件特性时这是不可避免的。

规则 11.4（建议）：　不应在某类型对象指针和其他不同类型对象指针之间进行强制转换。

如果新的指针类型需要更严格的分配时这样的转换可能是无效的。

```
uint8_t    *    p1;
uint32_t   *    p2;
p2 = (uint32_t *) p1;      /* Imcompatible alignment ? */
```

规则 11.5（强制）：　如果指针所指向的类型带有 const 或 volatile 限定符，那么移除限定符的强制转换是不允许的。

[未定义 39、40]

任何通过强制转换移除类型限定符的企图都是对类型限定符规则的违背。注意，这里所指的限定符与任何可以应用在指针本身的限定符不同。

```
uint16_t                    x;
uint16_t * const            cpi = &x;          /* const pointer */
uint16_t * const     *      pcpi ;             /* pointer to const pointer */
const uint16_t *     *      ppci ;             /* pointer to pointer to const */
uint16_t *           *      ppi;
const uint16_t       *      pci;               /* pointer to const */
volatile uint16_t    *      pvi;               /* pointer to volatile */
uint16_t             *      pi;
…
pi = cpi;                                      /* Compliant – no conversion
                                                  no cast required      */
pi = (uint16_t *)pci;                          /* Not compliant */
pi = (uint16_t *)pvi ;                         /* Not compliant */
ppi = (uint16_t *)pcpi ;                       /* Not compliant */
ppi = (uint16_t *)ppci ;                       /* Not compliant */
```

6.12 表达式

规则 12.1（建议）：　不要过分依赖 C 表达式中的运算符优先规则

括号的使用除了可以覆盖缺省的运算符优先级以外，还可以用来强调所使用的运算符。

使用相当复杂的 C 运算符优先级规则很容易引起错误，那么这种方法就可以帮助避免这样的错误，并且可以使得代码更为清晰可读。然而，过多的括号会分散代码使其降低了可读性。

下面的方针给出了何时使用括号的建议：

- 赋值运算符的右手操作数不需要使用括号，除非右手端本身包含了赋值表达式：

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

- 一元运算符的操作数不需要使用括号：

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

- 否则，二元和三元运算符的操作数应该是 cast-expressions（见 6.3.4 节 ISO 9899:1990 [2]），除非表达式中所有运算符是相同的。

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

- 即使所有运算符都是相同的，也可以使用括号控制运算的次序。某些运算符（如，加法和乘法）在代数学上结合律的，而在 C 中未必如此。类似地，涉及混合类型的整数运算（许多规则不允许）因为整数提升的存在可以产生不同的结果。下面的例子是按照 16 位的实现写成的，它描述了加法不是结合的以及表达式结构清晰的重要性：

```
uint16_t  a = 10;
uint16_t  b = 65535;
uint32_t  c = 0;
uint32_t  d;
d = (a + b) + c;            /* d is 9; a + b wraps modulo 65536 */
d = a + (b + c);            /* d is 65545 */
/* this example also deviates from several other rules */
```

注意，规则 12.5 是本规则的特例，它只能应用在逻辑运算符（&& 和 ||）上。

规则 12.2（强制）：　表达式的值在标准所允许的任何运算次序下都应该是相同的。

[未指定 7—9；未定义 18]

除了少数运算符（特别地，函数调用运算符 ( )、&&、||、?：和 ，（逗号） ）之外，子表达式所依据的运算次序是未指定的并会随时更改。这意味着不能信任子表达式的运算次序，特别不能信任可能会发生副作用（side effect）的运算次序。在表达式运算中的某些点上，如果能保证所有先前的副作用都已经发生，那么这些点称为“序列点（sequence point）”。序列点和副作用的描述见 ISO 9899:1990 [2]的 5.1.2.3 节、6.3 节和 6.6 节。

注意，运算次序的问题不能使用括号来解决，因为这不是优先级的问题。

下面的条款告诉我们对运算次序的依赖是如何发生的，并由此帮助我们采纳本规则。

- **自增或自减运算符**

做为能产生错误的例子，考虑

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

根据 b[i] 的运算是先于还是后于 i ++ 的运算，表达式会产生不同的结果。把增值运算做为单独的语句，可以避免这个问题。那么：

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

- **函数参数**

函数参数的运算次序是未指定的。

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

根据函数的两个参数的运算次序不同，表达式会给出不同的结果。

- **函数指针**

如果函数是通过函数指针调用的，那么函数标识符和函数参数运算次序是不可信任的。

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

- **函数调用**

函数在被调用时可以具有附加的作用（如，修改某些全局数据）。可以通过在使用函数的表达式之前调用函数并为值采用临时变量的方法避免对运算次序的依赖。

例如

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

可以写成

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

做为可以产生错误的例子，考虑下面的表达式，它从堆栈中取出两个值，从第一个值中减去第二个值，再把结果放回栈中：

```
push ( pop () – pop () );
```

根据哪一个 pop () 函数先进行计算（因为 pop()具有副作用）会产生不同的结果。

- **嵌套的赋值语句**

表达式中嵌套的赋值可以产生附加的副作用。不给这种能导致对运算次序的依赖提供任何机会的最好做法是，不要在表达式中嵌套赋值语句。

例如，下面的做法是不赞成的：

```
x = y = y = z / 3;
x = y = y++;
```

- **volatile 访问**

类型限定符 volatile 是 C 提供的，用来表示那些其值可以独立于程序的运行而自由更改的对象（例如输入寄存器）。对带有 volatile 限定类型的对象的访问可能改变它的值。C 编译器不会优化对 volatile 的读取，而且，据 C 程序所关心的，对 volatile 的读取具有副作用（改变 volatile 的值）。

做为表达式的一部分通常需要访问 volatile 数据，这意味着对运算次序的依赖。建议对 volatile 的访问尽可能地放在简单的赋值语句中，如：

```
volatile   uint16_t  v;
```

```
/* … */
x = v;
```

本规则讨论了带有副作用的运算次序问题。要注意子表达式的运算次数同样会带来问题，本规则没有提及。这是函数调用的问题，其中函数是以宏实现的。例如，考虑下面的函数宏及其调用：

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

当 a > b 时，该定义计算了两次第一个参数而在 a <= b 时只计算了一次。这样，宏调用根据 i 和 j 的值，对 i 增加了一次或两次。

应该说明的是，比如那些由浮点的四舍五入引起的量级依赖（magnitude-dependent）的作用也没有在这里涉及。尽管可能发生副作用的运算次序是未定义的，运算结果在另一方面是良好定义的并被表达式的结构所控制。在下面的例子中，f1 和 f2 是浮点变量；F3、F4 和 F5 代表浮点类型的表达式。

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

加法运算的次序由括号的位置决定，至少表面如此。即，首先 F4 的值加上 F5 然后加上 F3，给出 f1 的值。假定 F3、F4 和 F5 没有副作用，那么它们的值独立于它们被计算的次序。然而，赋给 f1 和 f2 的值不能保证是相同的，因为浮点的四舍五入后紧接加法的运算将依赖于被加的值。

规则 12.3（强制）：　　不能在具有副作用的表达式中使用 sizeof 运算符。

C 当中存在的一个可能的编程错误是为一个表达式使用了 sizeof 运算符并期望计算表达式。然而表达式是不会被计算的：sizeof 只对表达式的类型有用。为避免这样的错误，sizeof 不能用在具有副作用的表达式中，因为此时其副作用不会发生。例如：

```
int32_t    i;
int32_t    j;
j = sizeof (i = 1234);
    /* j is set to the sizeof the type of i which is an int */
    /* i is not set to 1234 */
```

规则 12.4（强制）：　　逻辑运算符 && 或 || 的右手操作数不能包含副作用。

C 当中存在这样的情况，表达式的某些部分不会被计算到。如果这些子表达式具有副作用，那么副作用可能会发生也可能不会发生，这依赖于其他子表达式的值。

可以导致这种问题的运算符是 &&、|| 和 ?: 。前两种情况（逻辑运算符）下，右手操作数的计算是有条件的，依赖于左手操作数的值。在 ?: 运算符情况下，或者第二个操作数被计算，或者第三个操作数被计算，却不会两者都被计算。两种逻辑运算符之一中，右手操作数的条件计算能轻易导致问题出现，如果程序员依赖副作用的发生。?: 运算符是被特殊用以在两个子表达式之间进行选择，因此导致错误的可能性较小。

例如：

```
if ( ishigh && ( x == i++ ) )        /* Not compliant */
if ( ishigh && ( x == f (x) ) )      /* Only acceptable if f(x) is
 known to have no side effects */
```

可以产生副作用的运算在 ISO 9899:1990 [2]的 5.1.2.3 节 中描述成 volatile 对象的访问、对象的修改、文件的修改或是执行某些运算的函数的调用，这里，函数所执行的这些运算可

以导致函数所运行的环境状态的改变。

规则 12.5（强制）：　逻辑 && 或 || 的操作数应该是 primary-expressions。

“Primary expressions”定义在 ISO 9899:1990 [2]的 6.3.1 节中。本质上它们或是单一的标识符，或是常量，或是括号括起来的表达式。本规则的作用是要求，如果操作数不是单一的标识符或常量，那么它必须被括起来。在这种情况下，括号对于代码的可读性和确保预期的行为都是非常重要的。如果表达式只由逻辑 && 序列组成或逻辑 | | 序列组成，就不需要使用括号。

例如：

```
if ( ( x == 0 ) && ishigh )        /* make x == 0 primary */
if ( x || y || z )                 /* exception allowed, if x, y and z are Boolean */
if ( x || ( y && z ) )             /* make y && z primary */
if ( x && ( !y ) )                 /* make !y primary */
if ( ( is_odd (y) ) && x )         /* make call primary */
```

如果表达式只由逻辑 && 序列组成或逻辑 || 序列组成，就不需要使用括号。

```
if ( ( x > c1 ) && ( y > c2 ) && ( z > c3 ) )       /* compliant */
if ( ( x > c1 ) && ( y > c2 ) || (z > c3 ) )        /* not compliant */
if ( ( x > c1 ) && ( ( y > c2 ) || ( z > c3 ) ) )   /* compliant extra ( ) used */
```

注意，本规则是规则 12.1 的特例。

规则 12.6（建议）：　逻辑运算符（&&、|| 和 !）的操作数应该是有效的布尔数。有效布尔类型的表达式不能用做非逻辑运算符（&&、|| 和 !）的操作数

[Koenig 48]

逻辑运算符 &&、|| 和 ！很容易同位运算符 &、| 和 ~ 混淆。见术语表中的“Boolean expressions”。

规则 12.7（强制）：　位运算符不能用于基本类型（underlying type）是有符号的操作数上。

[实现 17－19]

位运算（~、<<、>>、&、^ 和 | ）对有符号整数通常是无意义的。比如，如果右移运算把符号位移动到数据位上或者左移运算把数据位移动到符号位上，就会产生问题。

基本类型的描述见 6.10 节。

规则 12.8（强制）：　移位运算符的右手操作数应该位于零和某数之间，这个数要小于左手操作数的基本类型的位宽。

[未定义 32]

例如，如果左移或右移运算的左手操作数是 16 位整型，那么要确保它移动的位数位于 0 和 15 之间。

基本类型的描述见节 6.10。

有多种确保遵循本规则的方法。对右手操作数来说，最简单的是使其为一个常数（其值可以静态检查）。使用无符号整型可以保证该操作数非负，那么只有其上限需要检查（在运行时动态检查或者通过代码复查）。否则这两种限制都要被检查。

```
u8a = (uint8_t) (u8a << 7);                 /* compliant */
u8a = (uint8_t) (u8a << 9);                 /* not compliant */
u16a = (uint16_t) ( (uint16_t) u8a << 9 );  /* compliant */
```

规则 12.9（强制）：　一元减运算符不能用在基本类型无符号的表达式上。

把一元减运算符用在基本类型为 unsigned int 或 unsigned long 的表达式上时，会分别产生类型为 unsigned int 或 unsigned long 的结果，这是无意义的操作。把一元减运算符用在无符号短整型的操作数上，根据整数提升的作用它可以产生有意义的有符号结果，但这不是好的方法。

基本类型的描述见节 6.10。

规则 12.10（强制）： 不要使用逗号运算符。

使用逗号运算符通常不利于代码的可读性，可以使用其他方法达到相同的效果。

规则 12.11（建议）： 无符号整型常量表达式的计算不应产生折叠（wrap-around）。

因为无符号整型表达式不会严格意义上地溢出，但会以模的方式产生折叠，因此任何无符号整型常量表达式的有效“溢出”将不会被编译器检测到。尽管在运行时（run-time）有很好的理由依赖由无符号整型提供的模运算，但在编译时（compile-time）计算常量表达式该理由就不那么明显了。因此任何发生折叠的无符号整型常量表达式都可能表示编程错误。

本规则同等地应用于翻译过程的所有阶段。对于在编译时计算所选择的常量表达式，编译器以这样的方式计算，即其计算结果与在目标机上的运算结果相同，除了条件预处理指令。对这样的指令，可以使用常见的算术运算法则（见 ISO 9899:1990 [2]中 6.4 节），但是 int 和 unsigned int 的行为会被分别替代成好像它们是 long 或 unsigned long 一样。

例如，在 int 类型为 16 位、long 类型为 32 位的机器上：

```
#define   START   0x8000
#define   END     0xFFFF
#define   LEN     0x8000
#if ( (START + LEN ) > END )
#error Buffer Overrun    /* OK because START and LEN are unsigned long */
#endif
#if ( ( ( END – START ) – LEN ) < 0 )
    #error Buffer Overrun
                         /* Not OK: subtraction result wraps around to 0xFFFFFFFF */
#endif
/* contrast the above START + LEN with the following */
if ( ( START + LEN ) > END )
{
    error ( “Buffer overrun “ );
                         /* Not OK: START + LEN wraps around to 0x0000 due to
unsigned
                              int arithmetic */
}
```

规则 12.12（强制）： 不应使用浮点数的基本（underlying）的位表示法（bit representation）

[未指定 6；实现 20]

浮点数的存储方法可以根据编译器的不同而不同，因此不应使用直接依赖于存储方法的浮点操作。应该使用内置（in-built）的运算符和函数，它们对程序员隐藏了存储细节。

规则 12.13（建议）： 在一个表达式中，自增（++）和自减（- -）运算符不应同其他运算符混合在一起。

不建议使用同其他算术运算符混合在一起的自增和自减运算符，是因为：

- 它显著削弱了代码的可读性
- 它为语句引入了其他的副作用，可能存在未定义的行为

把这些操作同其他算术操作隔离开是比较安全的。

例如，下面的语句是不适合的：

```
u8a = ++u8b + u8c--;     /* Not compliant */
```

下面的序列更为清晰和安全：

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

6.13 控制语句表达式

规则 13.1（强制）：　赋值运算符不能使用在产生布尔值的表达式上。

[Koenig 6]

任何被认为是具有布尔值的表达式上都不能使用赋值运算。这排除了赋值运算符的简单与复杂的使用形式，其操作数是具有布尔值的表达式。然而，它不排除把布尔值赋给变量的操作。

如果布尔值表达式需要赋值操作，那么赋值操作必须在操作数之外分别进行。这可以帮助避免“=”和“= =”的混淆，帮助我们静态地检查错误。

见术语表中的“Boolean expressions”。

例如：

```
x = y;
if (x != 0)
{
    foo ();
}
```

不能写成：

```
if ( ( x = y ) != 0 )        /* Boolean by context */
{
    foo ();
}
```

或者更坏的：

```
if (x = y)
{
    foo ();
}
```

规则 13.2（建议）：　数的非零检测应该明确给出，除非操作数是有效的布尔类型。

当要检测一个数据不等于零时，该测试要明确给出。本规则的例外是该数据代表布尔类型的值，虽然在 C 中布尔数实际上也是整数。本规则的着眼点是在代码的清晰上，给出整数和逻辑数之间的清晰划分。

例如，如果 x 是个整数，那么：

```
if ( x != 0)      /* Correct way of testing x is non-zero */
if ( y )          /* Not compliant, unless y is effectively Boolean data (e.g. a flag). */
```

见术语表中的“Boolean expressions”。

规则 13.3（强制）：　浮点表达式不能做相等或不等的检测。

这是浮点类型的固有特性，等值比较通常不会计算为 true，即使期望如此。而且，这种比较行为不能在执行前做出预测，它会随着实现的改变而改变。例如，下面代码中的测试结果就是不可预期的：

```
float32_t       x,    y;
/* some calculations in here */
if ( x == y)     /* not compliant */
    { /* … */ }
if (x == 0.0f)   /* not compliant */
```

间接的检测同样是有问题的，在本规则内也是禁止的。例如：

```
if ( ( x <= y ) && ( x >= y ) )
    { /* … */ }
```

为了获得确定的浮点比较，建议写一个实现比较运算的库。这个库应该考虑浮点的粒度（FLT_EPSILON）以及参与比较的数的量级。见规则 13.4 和规则 20.3。

规则 13.4（强制）：　for 语句的控制表达式不能包含任何浮点类型的对象。

控制表达式可能会包含一个循环计数器，检测其值以决定循环的终止。浮点变量不能用于此目的。舍入误差和截取误差会通过循环的迭代过程传播，导致循环变量的显著误差，并且在进行检测时很可能给出不可预期的结果。例如，循环执行的次数可以随着实现的改变而改变，也是不可预测的。见规则 13.3。

规则 13.5（强制）：　for 语句的三个表达式应该只关注循环控制。

for 语句的三个表达式都给出时它们应该只用于如下目的：

第一个表达式　初始化循环计数器 （例子中的 i）

第二个表达式　应该包含对循环计数器（i）和其他可选的循环控制变量的测试

第三个表达式　循环计数器（i）的递增或递减

规则 13.6（强制）：　for 循环中用于迭代计数的数值变量不应在循环体中修改。

不能在循环体中修改循环计数器。然而可以修改表现为逻辑数值的其他循环控制变量。例如，指示某些事情已经完成的标记，然后在 for 语句中测试。

```
flag = 1;
for ( i = 0; ( i < 5 ) && (flag == 1 ); i++)
{
    /* … */
    flage = 0;       /* Compliant – allows early termination of loop */
    i = i + 3;       /* Not compliant – altering the loop counter */
}
```

规则 13.7（强制）：　不允许进行结果不会改变的布尔运算。

如果布尔运算产生的结果始终为“true”或始终为“false”，那么这很可能是编程错误。

```
enum ec { RED, BLUE, GREEN }        col;
…
if (u16a < 0)                       /* Not compliant – u16a is always >= 0 */
…
if (u16a <= 0xffff)                 /* Not compliant – always true */
…
if (s8a < 130)                      /* Not compliant – always true */
…
if ( ( s8a < 10 ) && ( s8a > 20) )  /* Not compliant – always false */
…
if ( ( s8a < 10 ) || ( s8a > 5) )   /* Not compliant – always true */
…
if ( col <= GREEN )                 /* Not compliant – always true */
…
if (s8a > 10)
{
    if (s8a > 5)                    /* Not compliant – s8a is not volatile */
    {
    }
}
```

6.14 控制流

规则 14.1（强制）：　不能有不可到达（unreachable）的代码。

本规则是针对那些在任何环境中都不能到达的代码，这些代码在编译时就能被标识出不可到达。规则排除了可以到达但永远不会执行的代码（如，保护性编程代码（defensive programming）　）。

如果从相关的入口到某部分代码之间不存在控制流路径，那么这部分代码就是不可到达的。例如，在无条件控制转移代码后的未标记代码就是不可到达的：

```
switch ( event )
{
case   E_wakeup:
    do_wakeup ();
    break;              /* unconditional control transfer */
    do_more ();         /* Not compliant – unreachable code */
    /* … */
default:
    /* … */
    break;
}
```

对整个函数来说，如果不存在调用它的手段，那么这个函数将是不可到达的。

规则 14.2（强制）：　所有非空语句（non-null statement）应该：

a)　不管怎样执行都至少有一个副作用（side-effect），或者

b)　可以引起控制流的转移

任何语句（非空语句），如果既没有副作用也不引起控制流的改变，通常就会指示出编程错误，因此要进行对这样语句的静态检测。例如，下面的语句在执行时不一定带有副作用：

```
/* assume uint16_t        x;
   and       uint16_t       i; */
…
x >= 3u;                /* not compliant: x is compared to 3,
                           and the answer is discarded */
```

注：“null statement”和“side effect”分别定义在 ISO 9899:1990 [2] 中节 6.6.3 和 5.1.2.3。

规则 14.3（强制）：　在预处理之前，空语句只能出现在一行上；其后可以跟有注释，假设紧跟空语句的第一个字符是空格。

通常不会故意包含空语句，但是在使用它们的地方，它们应该出现在它们本身的行上。空语句前面可以有空格以保持缩进的格式。如果一条注释跟在空语句的后面，那么至少要有一个空格把空语句和注释分隔开来。需要这样起分隔作用的空格是因为它给读者提供了重要的视觉信息。遵循本规则使得静态检查工具能够为与其他文本出现在一行上的空语句提出警告，因为这样的情形通常表示编程错误。例如：

```
while ( ( port & 0x80 ) == 0 )
{
    ;   /* wait for pin – Compliant */
    /* wait for pin */   ;   /* Not compliant, comment before ; */
    ;/* wait for pin – Not compliant, no white-space char after ; */
}
```

规则 14.4（强制）：　不应使用 goto 语句。

规则 14.5（强制）：　不应使用 continue 语句。

规则 14.6（强制）：　对任何迭代语句至多只应有一条 break 语句用于循环的结束。

这些规则关心的是良好的编程结构。循环中允许有一条 break 语句，因为这允许双重结果的循环或代码优化。

规则 14.7（强制）：　一个函数在其结尾应该有单一的退出点。

这是由 IEC 61508 良好的编程格式要求的。

规则 14.8（强制）：　组成 switch、while、do...while 或 for 结构体的语句应该是复合语句。

组成 switch 语句或 while、do ... while 或 for 循环结构体的语句应该是复合语句（括在大括号里），即使该复合语句只包含一条语句。

例如：

```
for ( i = 0 ; i < N_ELEMENTS ; ++i )
{
    buffer[i] = 0;          /* Even a single statement must be in braces */
}
while ( new_data_available )
```

```
    process_data ();          /* Incorrectly not enclosed in braces */
    service_watchdog ();      /* Added later but, despite the appearance
                                 (from the indent) it is actually not
                                 part of the body of the while statement,
                                 and is executed only afgter the loop has
                                 terminated */
```

注意，复合语句及其大括号的布局应该根据格式指南来确定。上述只是个例子。

规则 14.9（强制）： if（表达式）结构应该跟随有复合语句。else 关键字应该跟随有复合语句或者另外的 if 语句。

例如：

```
if ( test1 )
{
    x = 1 ;                   /* Even a single statement must be in braces */
}
else if ( test2 )
{
    x = 0;                    /* No need for braces in else if */
}
else
    x = 3;                    /* This was (incorrectly) not enclosed in braces */
    y = 2;                    /* This line was added later but, despite the appearance
                                 (from the indent) it is actually not part of the else,
                                 and is executed unconditionally */
```

注意，复合语句及其大括号的布局应该根据格式指南来确定。上述只是个例子。

规则 14.10（强制）： 所有的 if ... else if 结构应该由 else 子句结束。

不管何时一条 if 语句跟有一个或多个 else if 语句都要应用本规则；最后的 else if 必须跟有一条 else 语句。而 if 语句然后就是 else 语句的简单情况不在本规则之内。

对最后的 else 语句的要求是保护性编程（defensive programming）。else 语句或者要执行适当的动作，或者要包含合适的注释以说明为何没有执行动作。这与 switch 语句中要求具有最后一个 default 子句（规则 15.3）是一致的。

例如，下面的代码是简单的 if 语句：

```
if ( x > 0)
{
    log_error (3) ;
    x = 0 ;
}               /* else not needed */
```

而下面的代码描述了 if，else if 结构：

```
if ( x < 0 )
{
    log_error (3);
```

```
    x = 0;
}
else if (y < 0)
{
    x = 3;
}
else                    /* this else clause is required, even if the */
{                       /* programmer expects this will never be reached */
    /* no change in value of x */
}
```

6.15 switch 语句

C 当中 switch 语句的语法是软弱的，允许复杂的非结构化的行为。下面的文字描述了 MISRA-C 中定义的 switch 语句的标准语法。它们和与之相关的规则强制了简单一致的 switch 语句结构。

下面的语法规则是对 C 标准语法规则的补充，所有未在下面定义的语法规则都在 C 标准中做出了定义。

switch-statement:

switch (expression) **{** case-label-clause-list default-label-clause **}**

case-label-clause-list:

case-label case-clause$_{opt}$

case-label-clause-list case-label case-clause$_{opt}$

case-label:

case constant-expression **:**

case-clause:

statement-list$_{opt}$ **break;**

{ declaration-list$_{opt}$ statement-list$_{opt}$ **break; }**

default-label-clause:

default-label default-clause

default-label:

default :

default-clause:

case-clause

and

statement:

/* labelled_statement removed */

compound_statement

expression_statement

selection_statement

iteration_statement

```
/* jump_statement removed or just restricted to return depending on other rules */
```

规则中还使用了下面的术语：

switch label 或者是 case 标签，或者是 default 标签

case clause 任何两个 switch label 之间的代码

default clause default 标签和 switch 语句结尾之间的代码

switch clause 或者是 case clause，或者是 default clause

switch 语句只应包含 switch label 和 switch clause 而没有其他代码。

规则 15.1（强制）： switch 标签只能用在当最紧密封闭（closely-enclosing）的复合语句是 switch 语句体的时候

case 或 default 标签的范围应该是做为 switch 语句体的复合语句。所有 case 子句和 default 子句应该具有相同的范围。

规则 15.2（强制）： 无条件的 break 语句应该终止每个非空的 switch 子句。

[Koenig 22-24]

每个 switch 子句中的最后一条语句应该是 break 语句，或者如果 switch 子句是复合语句，那么复合语句的最后一条语句应该是 break 语句。

规则 15.3（强制）： switch 语句的最后子句应该是 default 子句。

对最后的 default 子句的要求是出于保护性编程。该子句应该执行适当的动作，或者包含合适的注释以说明为何没有执行动作。

规则 15.4（强制）： switch 表达式不应是有效的布尔值。

见术语表中的“Boolean expressions”。

例如：

```
switch ( x == 0)          /* not compliant – effectively Boolean */
{
    …
```

规则 15.5（强制）： 每个 switch 语句至少应有一个 case 子句。

例如：

```
switch (x)
{
    uint8_t   var;        /* not compliant – decl before 1st case */
case 0:
    a = b;
    break;                /* break is required here */
case 1:                   /* empty clause, break not required */
case 2:
    a = c;                /* executed if x is 1 or 2 */
    if ( a == b )
    {
        case 3:           /* not compliant – case is not allowed here */
    }
```

```
        break;
    case 4:
        a = b;              /* not compliant – non empty drop through */
    case 5:
        a = c;
        break;
    default:                /* default clause is required */
        errorflag = 1;      /* should be non-empty if possible */
        break;              /* break is required here, in case a
                               future modification turns this into a
                               case clause                    */
    }
```

6.16 函数

规则 16.1（强制）： 函数定义不得带有可变数量的参数

[未指定 15；未定义 25、45、61、70-76]

本特性存在许多潜在的问题。用户不应编写使用可变数量参数的附加函数。这排除了 stdarg.h、va_arg、va_start 和 va_end 的使用。

规则 16.2（强制）： 函数不能调用自身，不管是直接还是间接的。

这意味着在安全相关的系统中不能使用递归函数调用。递归本身承载着可用堆栈空间过度的危险，这能导致严重的错误。除非递归经过了非常严格的控制，否则不可能在执行之前确定什么是最坏情况（worst-case）的堆栈使用。

规则 16.3（强制）： 在函数的原型声明中应该为所有参数给出标识符

出于兼容性、清晰性和可维护性的原因，应该在函数的原型声明中为所有参数给出名字。

规则 16.4（强制）： 函数的声明和定义中使用的标识符应该一致

规则 16.5（强制）： 不带参数的函数应当声明为具有 void 类型的参数

[Koenig 59-62]

函数应该声明为具有返回类型（见规则 8.2），如果函数不返回任何数据，返回类型为 void。类似地，如果函数不带参数，参数列表应声明为 void。例如函数 myfunc，如果既不带参数也不返回数据则应声明为：

```
void   myfunc ( void );
```

规则 16.6（强制）： 传递给一个函数的参数应该与声明的参数匹配。

[未定义 22]

这个问题可以通过使用函数原型完全避免——见规则 8.1。本规则被保留是因为编译器可能不会标记这样的约束错误。

规则 16.7（建议）： 函数原型中的指针参数如果不是用于修改所指向的对象，就应该声明为指向 const 的指针。

本规则会产生更精确的函数接口定义。const 限定应当用在所指向的对象而非指针，因为要保护的是对象本身。

例如：

```
void myfunc ( int16_t     *param1, const int16_t   *param2, int16_t     *param3 )
/*    param1 : Addresses an object which is modified – no const
      param2 : Addresses an object which is not modified – const required
      param3 : Addresses an object which is not modified – const missing          */
{
      *param1 = *param2 + *param3;
      return;
}
/* data at address param3 has not been changed, but this is not const therefore
      not compliant */
```

规则 16.8（强制）：　带有 non-void 返回类型的函数其所有退出路径都应具有显式的带表达式的 return 语句。

[未定义 43]

表达式给出了函数的返回值。如果 return 语句不带表达式，将导致未定义的行为（而且编译器不会给出错误）。

规则 16.9（强制）：　函数标识符的使用只能或者加前缀&，或者使用括起来的参数列表，列表可以为空。

[Koenig 24]

如果程序员写为：

```
if (f)       /* not compliant – gives a constant non-zero value which is the address of f – use
                  either f () or &f   */
{
      /* … */
}
```

那么就不会清楚其意图是要测试函数的地址是否为 NULL，还是执行函数 f ()的调用。

规则 16.10（强制）：　如果函数返回了错误信息，那么错误信息应该进行测试。

一个函数（标准库中的函数、第三方库函数、或者是用户定义的函数）能够提供一些指示错误发生的方法。这可以通过使用错误标记、特殊的返回数据或者其他手段。不管什么时候函数提供了这样的机制，调用程序应该在函数返回时立刻检查错误指示。

然而要注意到，相对于在函数完成后才检测错误的做法（见规则 20.3）来说，对函数输入值的检查是更为鲁棒的防止错误的手段。还要注意到，对 errno 的使用（为了返回函数的错误信息）是笨拙的并应该谨慎使用（见规则 20.5）。

6.17 指针和数组

规则 17.1（强制）：　指针的数学运算只能用在指向数组或数组元素的指针上。

[未定义 30]

对并非指向数组或数组元素的指针做整数加减运算（包括增值和减值）会导致未定义的行为。

规则 17.2（强制）：　指针减法只能用在指向同一数组中元素的指针上。

[未定义 31]

只有当两个指针指向（或至少好象是指向了）同一数组内的对象时，指针减法才能给出良好定义的结果。

规则 17.3（强制）：　　>、>=、<、<= 不应用在指针类型上，除非指针指向同一数组。

[未定义 33]

如果两个指针没有指向同一个对象，那么试图对指针做比较将导致未定义的行为。注意：允许指向超出数组尾部的元素，但对该元素的访问是禁止的。

规则 17.4（强制）：　　数组的索引应当是指针数学运算的唯一可允许的方式

数组的索引是指针数学运算的唯一可接受的方式，因为它比其他的指针操作更为清晰并由此具有更少的错误倾向。本规则禁止了指针数值的显式运算。数组索引只能应用在定义为数组类型的对象上。任何显式计算的指针值潜在地会访问不希望访问的或无效的内存地址。指针可以超出数组或结构的范围，或者甚至可以有效地指向任意位置。见规则 21.1。

```
void my_fn (uint8_t * p1, uint8_t p2[] )
{
    uint8_t    index = 0 ;
    uint8_t    *p3 ;
    uint8_t    *p4 ;
    *p1 = 0 ;
    p1 ++ ;                    /* not compliant – pointer increment */
    p1 = p1 + 5 ;              /* not compliant – pointer increment */
    p1[5] = 0 ;                /* not compliant – p1 was not declared as an array */
    p3 = &p1[5];               /* not compliant – p1 was not declared as an array */
    p2[0] = 0;
    index ++;
    index = index + 5;
    p2[index] = 0;             /* compliant */
    p4 = &p2[5];               /* compliant */
}
uint8_t    a1[16];
uint8_t    a2[16];
my_fn (a1, a2);
my_fn (&a1[4], &a2[4]);
uint8_t    a[10];
uint8_t    *p;
p = a;
* (p + 5) = 0;                /* not compliant */
p[5] = 0;                      /* compliant */
```

规则 17.5（建议）：　　对象声明所包含的间接指针不得多于 2 级

多于 2 级的间接指针会严重削弱对代码行为的理解，因此应该避免。

```
typedef    int8_t      * INTPTR;
```

```
struct     s {
     int8_t     * s1;       /* compliant */
     int8_t     * s2;       /* compliant */
     int8_t     * s3;       /* compliant */
};
struct     s *          ps1;       /* compliant */
struct     s **         ps2;       /* compliant */
struct     s ***        ps3;       /* not compliant */
int8_t     **    ( *pfunc1 ) ();       /* compliant */
int8_t     **    ( **pfunc2 ) ();      /* compliant */
int8_t     **    ( ***pfunc3 ) ();     /* not compliant */
int8_t     ***   ( ** pfunc4 ) ();     /* not compliant */
void       function    ( int8_t   *     par1,
                         int8_t   **    par2,
                         int8_t   ***   par3,                  /* not compliant */
                         INTPTR *   par4,
                         INTPTR *   const * const par5,  /* not compliant */
                         int8_t   *     par6[],
                         int8_t   **    par7[] )               /* not compliant */
{
     int8_t *          ptr1 ;
     int8_t **         ptr2 ;
     int8_t ***        ptr3 ;                  /* not compliant */
     INTPTR *      ptr4 ;
     INTPTR *      const * const ptr5 ;  /* not compliant */
     int8_t *          ptr6[10];
     int8_t **         ptr7[10];
}
```

类型解释：

- par1 和 ptr1 是指向 int8_t 的指针。
- par2 和 ptr2 是指向 int8_t 的指针的指针。
- par3 和 ptr3 是指向 int8_t 的指针的指针的指针。这是三级，因此不适合。
- par4 和 ptr4 扩展开是指向 int8_t 的指针的指针。
- par5 和 ptr5 扩展开是指向 int8_t 的指针的 const 指针的 const 指针。这是三级，因此不合适。
- par6 是指向 int8_t 的指针的指针，因为数组被转化成指向数组初始元素的指针。
- ptr6 是 int8_t 类型的指针数组。
- par7 是指向 int8_t 的指针的指针的指针，因为数组被转化成指向数组初始元素的指针。这是三级，因此不合适。

- ptr7 是指向 int8_t 的指针的指针数组。这是合适的。

规则 17.6（强制）：　自动存储对象的地址不应赋值给其他的在第一个对象已经停止存在后仍然保持的对象。

[未定义 9、26]

如果一个自动对象的地址赋值给其他的或者具有更大作用域的对象、或者静态对象、或者从一个函数返回的对象，那么在初始对象停止存在（其地址成为无效）时，包含地址的对象可能会延续存在。

例如：

```
int8_t      * foobar (void)
{
    int8_t      local_auto;
    return (&local_auto);      /* not compliant */
}
```

6.18 结构与联合

规则 18.1（强制）：　所有结构与联合的类型应该在转换单元（translation unit）的结尾是完善的。

[未定义 35]

结构或联合的完整声明应该包含在任何涉及该结构的转换单元之内。见 ISO 9899:1990 [2] 中 6.1.2.5 节关于对不完整类型的详细描述。

```
struct      tnode * pt;             /* tnode is incomplete at this point */
struct      tnode
{
    int     count;
    struct      tnode * left;
    struct      tnode * right;
};                                  /* type tnode is now complete */
```

规则 18.2（强制）：　对象不能赋值给重叠（overlapping）对象。

[未定义 34、35]

当两个对象创建后，如果它们拥有重叠的内存空间并把一个拷贝给另外一个时，该行为是未定义的。

规则 18.3（强制）：　不能为了不相关的目的重用一块内存区域。

本规则涉及了使用内存存储某些数据并在程序仍然运行期间的其他时刻使用同一块内存存储不相关数据的技术。很明显这依赖于两段不同的数据块，它们存在于程序运行期间不相连的时间段而且从来不会被同时请求。

对于安全相关的系统，不建议这样做，因为它会带来许多危险。比如：一个程序可能试图访问某个区域的某种类型的数据，而当时该区域正在存储其他类型的数据（如，由中断引起的）。这两种类型的数据在存储上可能不同而且可能会侵占其他数据。这样在每次切换使用时，数据可能不会正确初始化。这样的做法在并发系统中尤其是危险的。

然而要了解有时出于效率的原因可能会要求这样的存储共享。此时就非常需要采取检测手段以确保错误类型的数据永远不会被访问，确保数据始终被适当地初始化了以及确保对其

他部分数据（如，由分配不同引起的）的访问是不可能的。所采取的检测手段应该归纳为文档，并在违背本规则的背离内被证明是正确的。

可以使用联合或其他手段做到这一点。

注意，MISRA-C 不想对编译器翻译源代码的实际行为给出限制，因为用户通常对此没有有效的控制。所以，举例来说，编译器中的内存分配，不管是动态堆、动态堆栈或静态的，可能会仅仅因为轻微的代码变化就发生显著的改变，即使是在相同的优化级别上。还要注意的是，某些优化会合理地产生，甚至在用户没有做出这样的请求时。

规则 18.4（强制）：　不要使用联合。

[实现 27]

规则 18.3 禁止为不相关的目的重用内存区域，然而，即使内存区域是为了相关目的重用的，仍然会存在数据被误解的风险。因此，提出本规则禁止针对任何目的而使用联合。

尽管如此，要认识到在某些情况下需要谨慎地使用联合以构建有效率的实现。发生这些情况时，如果所有相关的定义实现的行为被归纳为文档了，那么对本规则的违背也是可以接受的。实践中，可以在设计文档内指出编译器手册的实现章节。与此有关的实现行为的种类是：

- 填充——在联合的结尾插入了多少填充
- 排列——在联合中排列了多少结构成员
- 存储次序（endianess）——数据字中最有效的字节是存储在最低内存地址还是最高地址
- 位次序（bit-order）——字节中的位是如何计数的以及如何分配在位域中的

对如下情况违背规则是可以接受的：（a）数据的打包和解包，如在发送和接收消息时；（b）实现变量录取（variant records），假设变量是由通用的域（common fields）区分的。不带区分器（differentiator）的变量录取在任何情况下都是不合适的。

数据打包和解包

在本例中，使用联合访问一个 32 位数据字的字节以存储从高字节优先的网络上接收到的字节。这种特殊假设：

- uint32_t 类型占据 32 位
- uint8_t 类型占据 8 位
- 实现把数据字的最高有效字节存储在最低的内存地址

实现接收和打包的代码如下：

```
typedef    union {
        uint32_t   word;
        uint8_t    bytes[4];
} word_msg_t;
uint32_t  read_word_big_endian (void)
{
        word_msg_t    tmp;
        tmp.bytes[0] = read_byte ();
        tmp.bytes[1] = read_byte ();
        tmp.bytes[2] = read_byte ();
        tmp.bytes[3] = read_byte ();
```

```
    return (tmp.word);
}
```

值得注意的是上面的函数体可以写成如下可移植的形式：

```
uint32_t  read_word_big_endian (void)
{
    uint32_t  word;
    word =          ( ( uint32_t ) read_byte () ) << 24;
    word = word | ( ( ( uint32_t ) read_byte () ) << 16 );
    word = word | ( ( ( uint32_t ) read_byte () ) << 8 );
    word = word |   ( ( uint32_t ) read_byte () ) ;
    return (word);
}
```

不幸的是，面临可移植的实现时，大多数编译器产生的代码远非有效率的。当程序对高执行速度和低内存使用的要求胜于移植性时，使用联合的实现是可以考虑的。

变量录取

联合通常用于实现变量录取。每个变量享有共同的域并具有其特有的附加域。本例基于 CAN 总线校准协议（CAN Calibration Protocal， CCP），其中每个发送给 CCP 客户端的 CAN 消息共享两个通用的域，每个域占一个字节。其后最多可附加 6 个字节，这些字节的解释依赖于存储在第一个字节中的消息类型。

这个特定实现依赖于如下假设：

- uint16_t 类型占据 16 位
- uint8_t 类型占据 8 位
- 排列和打包的规则是，uint8_t 和 uint16_t 类型的结构成员之间不存在间隙

为了简化，本例中只考虑两种消息类型。这里给出的代码是不完整的，只是用来描述变量录取而不是做为 CCP 的实现模块。

```
/* The fields common to all CCP messages */
typedef   struct {
    uint8_t    msg_type;
    uint8_t    sequence_no;
} ccp_common_t;
/* CCP connect message */
typedef   struct {
    ccp_common_t      common_part;
    uint16_t          station_to_connect;
} ccp_connect_t;
/* CCP disconnect message */
typedef   struct {
    ccp_common_t      common_part;
    uint8_t    disconnect_command;
    uint8_t    pad;
```

```
        uint16_t   station_to_disconnect;
} ccp_disconnect_t;
/* The variant */
typedef    union {
        ccp_common_t       common;
        ccp_connect_t      connect;
        ccp_disconnect_t   disconnect;
} ccp_message_t;
void   process_ccp_message (ccp_message_t  *msg)
{
        switch (msg->common.msg_type)
        {
        case Ccp_connect:
                if (MY_STATION == msg->connect.station_to_connect)
                {
                        ccp_connect ();
                }
                break;
        case Ccp_disconnect:
                if (MY_STATION == msg->disconnect.station_to_disconnect)
                {
                        if (PERM_DISCONNECT == msg->disconnect.disconnect_command)
                        {
                                ccp_disconnect ();
                        }
                }
                break;
        default:
                break;      /* ignore unknown commands */
        }
}
```

6.19 预处理指令

规则 19.1（建议）：　　文件中的#include 语句之前只能是其他预处理指令或注释。

代码文件中所有#include 指令应该成组放置在接近文件顶部的位置。本规则说明，文件中可以优先#include 语句放置的只能是其他预处理指令或注释。

规则 19.2（建议）：　　#include 指令中的头文件名字里不能出现非标准字符。

[未定义 14]

如果在头文件名字预处理标记的 < 和 > 限定符或 ” 和 ” 限定符之间使用了 ‘ ，\ ，

或 /* 字符，该行为是未定义的。

规则 19.3（强制）：　#include 预处理指令应该跟随<filename>或”filename”序列。

[未定义 48]

例如，如下语句是允许的：

```
#include  “ filename.h”
#include  <filename.h>
#define  FILE  “filename.h”
#include  FILE
```

规则 19.4（强制）：　C 的宏只能扩展为用大括号括起来的初始化、常量、小括号括起来的表达式、类型限定符、存储类标识符或 do-while-zero 结构。

[Koenig 82-84]

这些是宏当中所有可允许使用的形式。存储类标识符和类型限定符包括诸如 extern、static 和 const 这样的关键字。使用任何其他形式的#define 都可能导致非预期的行为，或者是非常难懂的代码。

特别的，宏不能用于定义语句或部分语句，除了 do-while 结构。宏也不能重定义语言的语法。宏的替换列表中的所有括号，不管哪种形式的 ()、{}、[] 都应该成对出现。

do-while-zero 结构（见下面的例子）是在宏语句体中唯一可接受的具有完整语句的形式。do-while-zero 结构用于封装语句序列并确保其是正确的。注意：在宏语句体的末尾**必须**省略分号。

例如：

```
/* The following are compliant */
#define  PI   3.14159F                  /* Constant */
#define  XSTAL   10000000               /* Constant */
#define  CLOCK  (XSTAL / 16)            /* Constant expression */
#define  PLUS2(X)    ( (X) + 2 )        /* Macro expanding to expression */
#define  STOR    extern                 /* storage class specifier */
#define  INIT(value)   { (value), 0, 0 }   /* braced initialiser */
#define  READ_TIME_32 () \
    do  { \
        DISABLE_INTERRUPTS (); \
        time_now = (uint32_t) TIMER_HI << 16; \
        time_now = time_now | (uint32_t) TIMER_LO; \
        ENABLE_INTERRUPTS (); \
    } while (0)                         /* example of do-while-zero */
/* the following are NOT compliant */
#define  int32_t   long                 /* use typedef instead */
#define  STARTIF  if (                  /* unbalanced () and language redefinition */
```

规则 19.5（强制）：　宏不能在块中进行 #define 和 #undef。

C 语言中，尽管在代码文件中的任何位置放置#define 或#undef 是合法的，但把它们放在块中会使人误解为好像它们存在于块作用域。

通常，#define 指令要放在接近文件开始的地方，在第一个函数定义之前。而#undef 指令

通常不一定需要（见规则 19.6）。

规则 19.6（强制）：　不要使用#undef。

通常，#undef 是不需要的。当它出现在代码中时，能使宏的存在或含义产生混乱。

规则 19.7（建议）：　函数的使用优先选择函数宏（function-like macro）。

[Koenig 78-81]

由于宏能提供比函数优越的速度，函数提供了一种更为安全和鲁棒的机制。在进行参数的类型检查时尤其如此。函数宏的问题在于它可能会多次计算参数。

规则 19.8（强制）：　函数宏的调用不能缺少参数

[未定义 49]

这是一个约束错误，但是预处理器知道并忽略此问题。函数宏中的每个参数的组成必须至少有一个预处理标记，否则其行为是未定义的。

规则 19.9（强制）：　传递给函数宏的参数不能包含看似预处理指令的标记。

[未定义 50]

如果任何参数的行为类似预处理指令，使用宏替代函数时的行为将是不可预期的。

规则 19.10（强制）：　在定义函数宏时，每个参数实例都应该以小括号括起来，除非它们做为#或##的操作数。

[Koenig 78-81]

函数宏的定义中，参数应该用小括号括起来。例如一个 abs 函数可以定义成：

```
#define    abs (x)    ( ( (x) >= 0 ) ? (x) : -(x) )
```

不能定义成：

```
#define    abs (x)    ( ( (x) >= 0 ) ? x : -x )
```

如果不坚持本规则，那么当预处理器替代宏进入代码时，操作符优先顺序将不会给出要求的结果。

考虑前面第二个不正确的定义被替代时会发生什么：

```
z = abs ( a – b );
```

将给出如下结果：

```
z = ( ( a – b >= 0 ) ? a – b : -a – b );
```

子表达式 – a - b 相当于 (-a)-b，而不是希望的 –(a-b)。把所有参数都括进小括号中就可以避免这样的问题。

规则 19.11（强制）：　预处理指令中所有宏标识符在使用前都应先定义，除了#ifdef 和#ifndef 指令及 defined()操作符。

如果试图在预处理指令中使用未经定义的标识符，预处理器有时不会给出任何警告但会假定其值为零。#ifdef、#ifndef 和 defined()用来测试宏是否存在并由此进行排除。

例如：

```
#if x < 0        /* x assumed to be zero if not defined */
```

在标识符被使用之前要考虑使用#ifdef 进行测试。

注意，预处理标识符可以使用#define 指令来定义也可以在编译器调用所指定的选项中定义。然而更多的是使用#define 指令。

规则 19.12（强制）：　在单一的宏定义中最多可以出现一次 # 或 ## 预处理器操作符。

[未指定 12]

与 # 或 ## 预处理器操作符相关的计算次序如果未被指定则会产生问题。为避免该问题，在单一的宏定义中只能使用其中一种操作符（即，一个 #、或一个 ##、或都不用）。

规则 19.13（建议）： 不要使用# 或 ## 预处理器操作符。

[未指定 12]

与 # 或 ## 预处理器操作符相关的计算次序如果未被指定则会产生问题。编译器对这些操作符的实现是不一致的。为避免这些问题，最好不要使用它们。

规则 19.14（强制）： defined 预处理操作符只能使用两种标准形式之一。

[未定义 47]

defined 预处理操作符的两种可允许的形式为：

```
defined (identifier)
defined identifier
```

任何其他的形式都会导致未定义的行为，比如：

```
#if  defined (X > Y)    /* not compliant – undefined behaviour */
```

在 #if 或 #elif 预处理指令的扩展中定义的标记也会导致未定义的行为，应该避免。如：

```
#define  DEFINED    defined
#if  DEFINED (X)    /* not compliant – undefined behaviour */
```

规则 19.15（强制）： 应该采取防范措施以避免一个头文件的内容被包含两次。

当转换单元（translation unit）包含了复杂层次的嵌套头文件时，会发生某头文件被包含多于一次的情形。这最多会导致混乱。如果它导致了多个定义或定义冲突，其结果将是未定义的或者是错误的行为。

多次包含一个头文件可以通过认真的设计来避免。如果不能做到这一点，就需要采取阻止头文件内容被包含多于一次的机制。通常的手段是为每个文件配置一个宏；当头文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内容。

例如，一个名为“ahdr.h”的文件可以组织如下：

```
#ifndef  AHDR_H
#define  AHDR_H
/* The following lines will be excluded by the preprocessor if the file is included
more than once */
…
#endif
```

或者可以使用下面的形式

```
#ifdef    AHDR_H
#error    Header file is already included
#else
#define  AHDR_H
/* The following lines will be excluded by the preprocessor if the file is included
more than once */
…
#endif
```

规则 19.16（强制）： 预处理指令在句法上应该是有意义的，即使是在被预处理器排除的情

况下。

当一段源代码被预处理指令排除时，每个被排除语句的内容被忽略直到出现一个 #else、#elif 或 #endif 指令（根据上下文内容）。如果其中一个被排除指令的组成形式不好（badly formed），编译器忽略它时不会给出任何警告，这将带来不幸的后果。

本规则要求所有预处理指令在句法上是有效的，即使它们出现在被排除的代码块中。

特别地，要确保 #else 和 #endif 指令后不要跟随除空格之外的任何字符。在强制执行这个 ISO 要求时编译器并非始终一致。

```
#define    AAA      2
...
int    foo (void)
{
      int    x = 0 ;
      ...
#ifndef    AAA
      x = 1 ;
#else1                        /* Not compliant */
      x = AAA ;
#endif
      ...
      retun x ;
}
```

规则 19.17（强制）：　所有的 #else、#elif 和 #endif 预处理指令应该同与它们相关的 #if 或 #ifdef 指令放在相同的文件中。

当语句块的包含和排除是被一系列预处理指令控制时，如果所有相关联的指令没有出现在同一个文件中就会产生混乱。本规则要求所有的预处理指令序列 #if / #ifdef ... #elif ... #else ... #endif 应该放在同一个文件中。遵循本规则会保持良好的代码结构并能避免维护性问题。

注意，这并不排除把所有这样的指令放在众多被包含文件中的可能性，只要与某一序列相关的所有指令放在一个文件中即可。

file.c

```
#define    A
...
#ifdef      A
...
#include    “ file1.h”
#
#endif
...
#if    1
#include  “ file2.h”
...
EOF
```

file1.h

```
#if  1
...
#endif              /* Compliant */
EOF
```

file2.h

```
...
#endif              /* Not compliant */
```

6.20 标准库

规则 20.1（强制）：　标准库中保留的标识符、宏和函数不能被定义、重定义或取消定义。

[未定义 54、57、58、62]

通常 #undef 一个定义在标准库中的宏是件坏事。同样不好的是，#define 一个宏名字，而该名字是 C 的保留标识符或者标准库中做为宏、对象或函数名字的 C 关键字。例如，存在一些特殊的保留字和函数名字，它们的作用为人所熟知，如果对它们重新定义或取消定义就会产生一些未定义的行为。这些名字包括 defined、__LINE__、__FILE__、__DATE__、__TIME__、__STDC__、errno 和 assert。

#undef 的使用也可以参见规则 19.6。

ISO C 的保留标识符在本文档中参见 7.1.3 节和 7.13 节关于 ISO 9899 :1990 [2]，同时也应该被编译器的编写者写入文档。通常，所有以下划线开始的标识符都是保留的。

规则 20.2（强制）：　不能重用标准库中宏、对象和函数的名字。

如果程序员使用了标准库中宏、对象或函数的新版本（如，功能增强或输入值检查），那么更改过的宏、对象或函数应该具有新的名字。这是用来避免不知是使用了标准的宏、对象或函数还是使用了它们的更新版本所带来的任何混淆。所以，举例来说，如果 sqrt 函数的新版本被写做检查输入值非负，那么这新版本不能命名为“sqrt”而应该给出新的名字。

规则 20.3（强制）：　传递给库函数的值必须检查其有效性。

[未定义 60、63；实现 45、47]

C 标准库中的许多函数根据 ISO 标准 [2] 并不需要检查传递给它们的参数的有效性。即使标准要求这样，或者编译器的编写者声明要这么做，也不能保证会做出充分的检查。因此，程序员应该为所有带有严格输入域的库函数（标准库、第三方库及自己定义的库）提供适当的输入值检查机制。

具有严格输入域并需要检查的函数例子为：

- math.h 中的许多数学函数，比如：

  负数不能传递给 sqrt 或 log 函数；

  fmod 函数的第二个参数不能为零

- toupper 和 tolower：当传递给 toupper 函数的参数不是小写字符时，某些实现能产生并非预期的结果（tolower 函数情况类似）
- 如果为 ctype.h 中的字符测试函数传递无效的值时会给出未定义的行为
- 应用于大多数负整数的 abs 函数给出未定义的行为

在 math.h 中，尽管大多数数学库函数定义了它们允许的输入域，但在域发生错误时它们的返回值仍可能随编译器的不同而不同。因此，对这些函数来说，预先检查其输入值的有效

性就变得至关重要。

程序员在使用函数时，应该识别应用于这些函数之上的任何的域限制（这些限制可能会也可能不会在文档中说明），并且要提供适当的检查以确认这些输入值位于各自域中。当然，在需要时，这些值还可以更进一步加以限制。

有许多方法可以满足本规则的要求，包括：

- 调用函数前检查输入值
- 设计深入函数内部的检查手段。这种方法尤其适应于实验室内开发的库，纵然它也可以用于买进的第三方库（如果第三方库的供应商声明他们已内置了检查的话）。
- 产生函数的“封装”（wrapped）版本，在该版本中首先检查输入，然后调用原始的函数。
- 静态地声明输入参数永远不会采取无效的值。

注意，在检查函数的浮点参数时（浮点参数在零点上为奇点），适当的做法是执行其是否为零的检查。这对规则 13.3 而言是可以接受的例外，不需给出背离。然而如果当参数趋近于零时，函数值的量级趋近无穷的话，仍然有必要检查其在零点（或其他任何奇点）上的容限，这样可以避免溢出的发生。

规则 20.4（强制）：　不能使用动态堆的内存分配

[未指定 19；未定义 91、92；实现 69；Koenig 32]

这排除了对函数 alloc、malloc、realloc 和 free 的使用。

涉及动态内存分配时，存在整个范围内的未指定的、未定义的和实现定义的行为，以及其他大量的潜在缺陷。动态堆内存分配能够导致内存泄漏、数据不一致、内存耗尽和不确定的行为。

注意，某些实现可能会使用动态堆内存的分配以实现其他函数（如库 string.h 中的函数）。如果这种情况发生，也需要避免使用这些函数。

规则 20.5（强制）：　不要使用错误指示 errno。

[实现 46；Koenig 73]

errno 做为 C 的简捷工具，在理论上是有用的，但在实际中标准没有很好地定义它。一个非零值可以指示问题的发生，也可以不用它指示；做为结果不应该使用它。即使对于那些已经良好定义了 errno 的函数而言，宁可在调用函数前检查输入值也不依靠 errno 来捕获错误（见规则 16.10）。

规则 20.6（强制）：　不应使用库<stddef.h>中的宏 offsetof。

[未定义 59]

当这个宏的操作数的类型不兼容或使用了位域时，它的使用会导致未定义的行为。

规则 20.7（强制）：　不应使用 setjmp 宏和 longjmp 函数

[未指定 14；未定义 64－67；Koenig 74]

setjmp 和 longjmp 允许绕过正常的函数调用机制，不应该使用。

规则 20.8（强制）：　不应使用信号处理工具<signal.h>

[未定义 68、69；实现 48－52；Koenig 74]

信号处理包含了实现定义的和未定义的行为。

规则 20.9（强制）：　在产品代码中不应使用输入/输出库<stdio.h>。

[未指定 2－5，16－18；未定义 77－89；实现 53－68]

这包含文件和 I/O 函数 fgetpos、fopen、ftell、gets、perror、remove、rename 和 ungetc。

流和文件 I/O 具有大量未指定的、未定义的和实现定义的行为。本文档中假定正常情况下嵌入式系统的产品代码中不需要它们。

如果产品代码中需要 stdio.h 中的任意特性，那么需要了解与此特性相关的某些问题。

规则 20.10（强制）： 不应使用库<stdlib.h>中的函数 atof、atoi 和 atol。

[未定义 90]

当字符串不能被转换时，这些函数具有未定义的行为。

规则 20.11（强制）： 不应使用库<stdlib.h>中的函数 abort、exit、getenv 和 system。

[未定义 93；实现 70－73]

正常情况下，嵌入式系统不需要这些函数，因为嵌入式系统一般不需要同环境进行通讯。如果一个应用中必需这些函数，那么一定要在所处环境中检查这些函数的实现定义的行为。

规则 20.12（强制）： 不应使用库<time.h>中的时间处理函数。

[未指定 22；未定义 97；实现 75、76]

包括 time、strftime。这个库同时钟有关。许多方面都是实现定义的或未指定的，如时间的格式。如果要使用 time.h 中的任一功能，那么必须要确定所用编译器对它的准确实现，并给出背离。

6.21 运行时错误

规则 21.1（强制）： 最大限度降低运行时错误必须要确保至少使用了下列方法之一：

a) 静态分析工具/技术；

b) 动态分析工具/技术；

c) 显式的代码检测以处理运行时故障

[未定义 19、26、94]

运行时检测是个问题，它不专属于 C，但 C 程序员需要加以特别的注意。这是因为 C 语言在提供任何运行时检测方面能力较弱。对于鲁棒的软件来说，动态检测是必需的，但 C 的实现不需要。因此 C 程序员需要谨慎考虑的问题是，在任何可能出现运行时错误的地方增加代码的动态检测。

当表达式仅仅由处在良好定义范围内的值组成时，倘若可以声称，对于所有处在定义范围内的值来说不会发生异常的话，运行时检测就不是必需的。如果使用了这样的声明，它应该与它所依赖的假设一起组织成文档。然而如果使用了这种方法，一定要小心的是，后续的代码改变可能会使原先的假设无效，或者要小心出于某种原因而改变了原先的假设。

下面的条款给出了在何处需要考虑提供动态检测的指导：

- **数学运算错误**

  它包括表达式运算错误，如溢出、下溢出、零除或移位时有效位的丢失。

  考虑整数溢出，注意，无符号的整数计算不会产生严格意义上的溢出（产生未定义的值），但是会产生值的折叠（wrap-around）（产生经过定义的但可能是错误的值）。

- **指针运算**

  确保在动态计算一个地址时，被计算的地址是合理的并指向某个有意义的地方。特别要保证指向一个结构或数组的内部，那么当指针增加或者改变后仍然指向同一个结构或数组。参见指针运算的限制——规则 17.1、17.2、17.4。

- **数组界限错误**

在使用数组索引元素前要确保它处于数组大小的界限之内

- **函数参数**

  见规则 20.3

- **引用指针的值（pointer dereferencing）**

  如果一个函数返回一个指针，而接下来该指针的值被引用，那么程序首先要检查指针不是 NULL。在一个函数内部，指出哪个指针可以保持 NULL 哪个指针不可以保持 NULL 是相对简单的事情。而跨越函数界限，尤其是在调用定义在其他文件或库中的函数时，这就困难得多。

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

用于最小化运行时错误的技术应该详细计划并写成文档，比如，在设计标准、测试计划、静态分析配置文件、代码检查清单中。

7 References

[1] MISRA Guidelines for the Use of the C Language In Vehicle Based Software, ISBN 0-9524156-9-0. Motor Industry Research Association. Nuneaton. April 1998

[2] ISO/IEC 9899:1990. Programming languages - C. International Organization for Standardization. 1990

[3] Hatton L... Safer C - Developing Software for High-integrity and Safety-critical Systems. ISBN 0-07-707640-0. McGraw-Hill, 1994

[4] ISO/IEC 9899:COR1:1995. Technical Corrigendum 1, 1995

[5] ISO/IEC 9899:AMD1:1995. Amendment 1. 1995

[6] ISO/IEC 9899:COR2:1996. 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 Organization for Standardization, 1999

[9] MISRA Development Guidelines for Vehicle Based Software, ISBN 0-9524156-0-7. Motor Industry Research Association. Nuneaton, November 1994

[10] CRR80, The Use of Commercial Off-the-Shelf (COTS) Software in Safety Related Applications. ISBN 0-7176-0984-7, HSE Books '

[11] ISO 9001:2000. Quality management systems - Requirements. International] Organization for Standardization. 2000

[12] ISO 90003:2004. Software engineering - Guidelines for the application of ISO 9001:2000 to computer software. International Organization for Standardization. 2004

[13] The TickIT Guide. Using ISO 9001:2000 for Software Quality Management System Construction, Certification and Continual Improvement, Issue 5, British Standards Institution. 2001

[14] Straker D.. C Style: Standards and Guidelines. ISBN 0-13-116898-3. Prentice Hall 1991

[15] Fenton N.E. and Pfleeger S.L., Software Metrics: A Rigorous and Practical Approach, 2nd Edition. ISBN 0-534-95429-1. PWS. 1998

[16] MISRA Report 5 Software Metrics. Motor Industry Research Association. Nuneaton. February 1995

[17] MISRA Report 6 Verification and Validation. Motor Industry Research Association. Nuneaton. February 1995

[18] Kemighan 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)

[19] Koenig A., C Traps and Pitfalls, ISBN 0-201 -17928-8, Addison-Wesley, 1988

[20] IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission, in 7 parts published between 1998 and 2000

[21] ANSI/IEEE Std 754, IEEE Standard for Binary Floating-Point Arithmetic, 1985

[22] ISO/IEC 10646:2003, Information technology - Universal Multiple-Octet Coded Character Set (UCS), International Organization for Standardization, 2003

Appendix A: Summary of rules

本附录给出第 6 节中所有规则的汇总。

环境

1.1 (req)	所有代码都必须遵照 ISO 9899:1990 “Programming languages - C”，由 ISO/IEC 9899/COR1:1995， ISO/IEC 9899/AMD1:1995， 和 ISO/IEC 9899/COR2:1996 修订。
1.2 (req)	不能有对未定义行为或未指定行为的依赖性。
1.3 (req)	多个编译器和/或语言只能在为语言/编译器/汇编器所适合的目标代码定义了通用接口标准时使用。
1.4 (req)	编译器/链接器要确保 31 个有效字符和大小写敏感能被外部标识符支持。
1.5 (adv)	浮点应用应该适应于已定义的浮点标准

语言扩展

2.1 (req)	汇编语言应该被封装并隔离。
2.2 (req)	源代码应该使用 /*…*/ 类型的注释。
2.3 (req)	字符序列 /* 不应出现在注释中。
2.4 (adv)	代码段不应被“注释掉”（comment out）。

文档

3.1 (req)	所有实现定义（implementation-defined）的行为的使用都应该文档化。
3.2 (req)	字符集和相应的编码应该文档化。
3.3 (adv)	应该确定、文档化和重视所选编译器中整数除法的实现。
3.4 (req)	所有#pragma 指令的使用应该文档化并给出良好解释。
3.5 (req)	如果做为其他特性的支撑，实现定义（implementation-defined）的行为和位域（bitfields）集合应当文档化。
3.6 (req)	产品代码中使用的所有库都要适应本文档给出的要求，并且要经过适当的验证。

字符集

4.1 (req)	只能使用 ISO C 标准中定义的 escape 序列。
4.2 (req)	不能使用三字母词（trigraphs）。

标识符

5.1 (req)	标识符（内部的和外部的）的有效字符不能多于 31。
5.2 (req)	具有内部作用域的标识符不应使用与具有外部作用域的标识符相同的名称，这会隐藏了外部标识符。
5.3 (req)	typedef 的名字应当是唯一的标识符。
5.4 (req)	标签（tag）名称必须是唯一的标识符。
5.5 (adv)	具有静态存储期的对象或函数标识符不能重用。
5.6 (adv)	一个命名空间中不应存在与另外一个命名空间中的标识符拼写相同的标识符，除了结构和联合的成员名字。
5.7 (adv)	不能重用标识符名字。

类型

6.1 (req) 单纯的 char 类型应该只用做存储和使用字符值。

6.2 (req) signed char 和 unsigned char 类型应该只用做存储和使用数字值。

6.3 (adv) 应该使用指示了大小和符号的 typedef 以代替基本类型。

6.4 (req) 位域只能被定义为 unsigned int 或 singed int 类型。

6.5 (req) unsigned int 类型的位域至少应该为 2 bits 长度。

常量

7.1 (req) 不应使用八进制常量（零除外）和八进制 escape 序列。

声明与定义

8.1 (req) 函数应当具有原型声明，且原型在函数的定义和调用范围内都是可见的。

8.2 (req) 不论何时声明或定义了一个对象或函数，它的类型都应显式声明。

8.3 (req) 函数的每个参数类型在声明和定义中必须是等同的，函数的返回类型也该是等同的。

8.4 (req) 如果对象或函数被声明了多次，那么它们的类型应该是兼容的。

8.5 (req) 头文件中不应有对象或函数的定义。

8.6 (req) 函数应该声明为具有文件作用域。

8.7 (req) 如果对象的访问只是在单一的函数中，那么对象应该声明在块范围内声明。

8.8 (req) 外部对象或函数应该声明在唯一的文件中。

8.9 (req) 具有外部链接的标识符应该具有准确的外部定义。

8.10 (req) 在文件范围内声明和定义的所有对象或函数应该具有内部链接，除非是在需要外部链接的情况下。

8.1l (req) static 存储类标识符具有内部链接的对象和函数的定义和声明。

8.12 (req) 当一个数组声明为具有外部链接，它的大小应该显式声明或者通过初始化进行隐式定义。

初始化

9.1 (req) 所有自动变量在使用前都应被赋值。

9.2 (req) 应该使用大括号以指示和匹配数组和结构的非零初始化构造。

9.3 (req) 在枚举列表中，“=”不能显式用于除首元素之外的元素上，除非所有的元素都是显式初始化的。

数值类型转换

10.1 (req) 下列条件成立时，整型表达式的值不应隐式转换为不同的基本类型：

a) 转换不是带符号的向更宽整数类型的转换，或者

b) 表达式是复杂表达式，或者

c) 表达式不是常量而是函数参数，或者

d) 表达式不是常量而是返回的表达式。

10.2 (req) 下列条件成立时，浮点类型表达式的值不应隐式转换为不同的类型：

a) 转换不是向更宽浮点类型的转换，或者

b) 表达式是复杂表达式，或者

c) 表达式是函数参数，或者

d) 表达式是返回表达式。

10.3 (req) 整型复杂表达式的值只能强制转换到更窄的类型且与表达式的基本类型具有相同的符号。

10.4 (req) 浮点类型复杂表达式的值只能强制转换到更窄的浮点类型。

10.5 (req) 如果位运算符 ~ 和 << 应用在基本类型为 unsigned char 或 unsigned short 的操作数，结果应该立即强制转换为操作数的基本类型。

10.6 (req) 后缀“U”应该用在所有 unsigned 类型的常量上。

指针类型转换

11.1 (req) 转换不能发生在函数指针和其他除了整型之外的任何类型指针之间。

11.2 (req) 对象指针和其他除整型之外的任何类型指针之间、对象指针和其他类型对象的指针之间、对象指针和 void 指针之间不能进行转换。

11.3 (adv) 不应在指针类型和整型之间进行强制转换

11.4 (adv) 不应在某类型对象指针和其他不同类型对象指针之间进行强制转换。

11.5 (req) 如果指针所指向的类型带有 const 或 volatile 限定符，那么移除限定符的强制转换是不允许的。

表达式

12.1 (adv) 不要过分依赖 C 表达式中的运算符优先规则

12.2 (req) 表达式的值在标准所允许的任何运算次序下都应该是相同的。

12.3 (req) 不能在具有副作用的表达式中使用 sizeof 运算符。

12.4 (req) 逻辑运算符 && 或 || 的右手操作数不能包含副作用。

12.5 (req) 逻辑 && 或 || 的操作数应该是 primary-expressions。

12.6 (adv) 逻辑运算符（&&、|| 和 !）的操作数应该是有效的布尔数。有效布尔类型的表达式不能用做非逻辑运算符（&&、|| 和 !）的操作数

12.7 (req) 位运算符不能用于基本类型（underlying type）是有符号的操作数上。

12.8 (req) 移位运算符的右手操作数应该位于零和某数之间，这个数要小于左手操作数的基本类型的位宽。

12.9 (req) 一元减运算符不能用在基本类型无符号的表达式上。

12.10 (req) 不要使用逗号运算符。

12.11 (adv) 无符号整型常量表达式的计算不应产生折叠（wrap-around）。

12.12 (req) 不应使用浮点数的基本（underlying）的位表示法（bit representation）

12.13 (adv) 在一个表达式中，自增（++）和自减（- -）运算符不应同其他运算符混合在一起。

控制语句表达式

13.1 (req) 赋值运算符不能使用在产生布尔值的表达式上。

13.2 (adv) 数的非零检测应该明确给出，除非操作数是有效的布尔类型。

13.3 (req) 浮点表达式不能做相等或不等的检测。

13.4 (req) for 语句的控制表达式不能包含任何浮点类型的对象。

13.5 (req) for 语句的三个表达式应该只关注循环控制。

13.6 (req) for 循环中用于迭代计数的数值变量不应在循环体中修改。

13.7 (req) 不允许进行结果不会改变的布尔运算。

控制流

14.1 (req) 不能有不可到达（unreachable）的代码。

14.2 (req) 所有非空语句（non-null statement）应该：

a) 不管怎样执行都至少有一个副作用（side-effect），或者

b) 可以引起控制流的转移

14.3 (req) 在预处理之前，空语句只能出现在一行上；其后可以跟有注释，假设紧跟空语句的第一个字符是空格。

14.4 (req) 不应使用 goto 语句。

14.5 (req) 不应使用 continue 语句。

14.6 (req) 对任何迭代语句至多只应有一条 break 语句用于循环的结束。

14.7 (req) 一个函数在其结尾应该有单一的退出点。

14.8 (req) 组成 switch、while、do...while 或 for 结构体的语句应该是复合语句。

14.9 (req) if（表达式）结构应该跟随有复合语句。else 关键字应该跟随有复合语句或者另外的 if 语句。

14.10 (req) 所有的 if ... else if 结构应该由 else 子句结束。

Switch 语句

15.1 (req) switch 标签只能用在当最紧密封闭（closely-enclosing）的复合语句是 switch 语句体的时候

15.2 (req) 无条件的 break 语句应该终止每个非空的 switch 子句。

15.3 (req) switch 语句的最后子句应该是 default 子句。

15.4 (req) switch 表达式不应是有效的布尔值。

15.5 (req) 每个 switch 语句至少应有一个 case 子句。

函数

16.1 (req) 函数定义不得带有可变数量的参数

16.2 (req) 函数不能调用自身，不管是直接还是间接的。

16.3 (req) 在函数的原型声明中应该为所有参数给出标识符

16.4 (req) 函数的声明和定义中使用的标识符应该一致

16.5 (req) 不带参数的函数应当声明为具有 void 类型的参数

16.6 (req) 传递给一个函数的参数应该与声明的参数匹配。

16.7 (adv) 函数原型中的指针参数如果不是用于修改所指向的对象，就应该声明为指向 const 的指针。

16.8 (req) 带有 non-void 返回类型的函数其所有退出路径都应具有显式的带表达式的 return 语句。

16.9(req) 函数标识符的使用只能或者加前缀&，或者使用括起来的参数列表，列表可以为空。

16.10 (req) 如果函数返回了错误信息，那么错误信息应该进行测试。

指针和数组

17.1 (req) 指针的数学运算只能用在指向数组或数组元素的指针上。

17.2 (req) 指针减法只能用在指向同一数组中元素的指针上。

17.3 (req) >、>=、<、<= 不应用在指针类型上，除非指针指向同一数组。

17.4 (req) 数组的索引应当是指针数学运算的唯一可允许的方式

17.5 (adv) 对象声明所包含的间接指针不得多于 2 级

17.6(req) 自动存储对象的地址不应赋值给其他的在第一个对象已经停止存在后仍然保持的对象。

结构与联合

18.1 (req) 所有结构与联合的类型应该在转换单元（translation unit）的结尾是完善的。

18.2 (req) 对象不能赋值给重叠（overlapping）对象。

18.3 (req) 不能为了不相关的目的重用一块内存区域。

18.4 (req) 不要使用联合。

预处理指令

19.1 (adv) 文件中的#include 语句之前只能是其他预处理指令或注释。

19.2 (adv) #include 指令中的头文件名字里不能出现非标准字符。

19.3 (req) #include 预处理指令应该跟随<filename>或”filename”序列。

19.4 (req) C 的宏只能扩展为用大括号括起来的初始化、常量、小括号括起来的表达式、类型限定符、存储类标识符或 do-while-zero 结构。

19.5 (req) 宏不能在块中进行 #define 和 #undef。

19.6 (req) 不要使用#undef。

19.7 (adv) 函数的使用优先选择函数宏（function-like macro）。

19.8 (req) 函数宏的调用不能缺少参数

19.9 (req) 传递给函数宏的参数不能包含看似预处理指令的标记。

19.10 (req) 在定义函数宏时，每个参数实例都应该以小括号括起来，除非它们做为#或##的操作数。

19.11 (req) 预处理指令中所有宏标识符在使用前都应先定义，除了#ifdef 和#ifndef 指令及 defined()操作符。

19.12 (req) 在单一的宏定义中最多可以出现一次 # 或 ## 预处理器操作符。

19.13 (adv) 不要使用# 或 ## 预处理器操作符。

19.14 (req) defined 预处理操作符只能使用两种标准形式之一。

19.15 (req) 应该采取防范措施以避免一个头文件的内容被包含两次。

19.16 (req) 预处理指令在句法上应该是有意义的，即使是在被预处理器排除的情况下。

19.17 (req) 所有的 #else、#elif 和 #endif 预处理指令应该同与它们相关的 #if 或 #ifdef 指令放在相同的文件中。

标准库

20.1 (req) 标准库中保留的标识符、宏和函数不能被定义、重定义或取消定义。

20.2 (req) 不能重用标准库中宏、对象和函数的名字。

20.3 (req) 传递给库函数的值必须检查其有效性。

20.4 (req)　不能使用动态堆的内存分配

20.5 (req)　不要使用错误指示 errno。

20.6 (req)　不应使用库<stddef.h>中的宏 offsetof。

20.7 (req)　不应使用 setjmp 宏和 longjmp 函数

20.8 (req)　不应使用信号处理工具<signal.h>

20.9 (req)　在产品代码中不应使用输入/输出库<stdio.h>。

20.10 (req)　不应使用库<stdlib.h>中的函数 atof、atoi 和 atol。

20.11 (req)　不应使用库<stdlib.h>中的函数 abort、exit、getenv 和 system。

20.12 (req)　不应使用库<time.h>中的时间处理函数。

运行时错误

21.1 (req)　最大限度降低运行时错误必须要确保至少使用了下列方法之一：

a)　静态分析工具/技术；

b)　动态分析工具/技术；

c)　显式的代码检测以处理运行时故障

Appendix B：MISRA-C :1998 到 MISRA-C :2004 规则映射

MISRA-C :1998	MISRA-C :2004	MISRA-C:2004 规则
1 (req)	1.1 (req)	所有代码都必须遵照ISO 9899:1990 "Programming languages - C"，由ISO/IEC 9899/COR1:1995，ISO/IEC 9899/AMD1:1995，和ISO/IEC 9899/COR2:1996修订。
1 (req)	1.2 (req)	不能有对未定义行为或未指定行为的依赖性。
1 (req)	2.2 (req)	源代码应该使用 /*…*/ 类型的注释。
1 (req)	3.1 (req)	所有实现定义（implementation-defined）的行为的使用都应该文档化。
2 (adv)	1.3 (req)	多个编译器和/或语言只能在为语言/编译器/汇编器所适合的目标代码定义了通用接口标准时使用。
3 (adv)	2.1 (req)	汇编语言应该被封装并隔离。
4 (adv)	21.1 (req)	最大限度降低运行时错误必须要确保至少使用了下列方法之一： a) 静态分析工具/技术； b) 动态分析工具/技术； c) 显式的代码检测以处理运行时故障
5 (req)	4.1 (req)	只能使用ISO C标准中定义的escape序列。
6 (req)	3.2 (req)	字符集和相应的编码应该文档化。
7 (req)	4.2 (req)	不能使用三字母词（trigraphs）。
8 (req)		已废除
9 (req)	2.3 (req)	字符序列 /* 不应出现在注释中。
10 (adv)	2.4 (adv)	代码段不应被"注释掉"（comment out）。
11 (req)	1.4 (req)	编译器/链接器要确保31个有效字符和大小写敏感能被外部标识符支持。
11 (req)	5.1 (req)	标识符（内部的和外部的）的有效字符不能多于31。
12 (adv)	5.5 (adv)	具有静态存储期的对象或函数标识符不能重用。
12 (adv)	5.6 (adv)	一个命名空间中不应存在与另外一个命名空间中的标识符拼写相同的标识符，除了结构和联合的成员名字。
12 (adv)	5.7 (adv)	不能重用标识符名字。
13 (adv)	6.3 (adv)	应该使用指示了大小和符号的typedef以代替基本类型。

MISRA-C :1998	MISRA-C :2004	MISRA-C:2004 规则
14 (req)	6.1 (req)	单纯的char类型应该只用做存储和使用字符值。
14 (req)	6.2 (req)	signed char和unsigned char类型应该只用做存储和使用数字值。
15 (adv)	1.5 (adv)	浮点应用应该适应于已定义的浮点标准
16 (req)	12.12 (req)	不应使用浮点数的基本（underlying）的位表示法（bit representation）
17 (req)	5.2 (req)	typedef的名字应当是唯一的标识符
18 (adv)		已废除
19 (req)	7.1 (req)	不应使用八进制常量（零除外）和八进制escape序列。
20 (req)		已废除
21 (req)	5.2 (req)	具有内部作用域的标识符不应使用与具有外部作用域的标识符相同的名称，这会隐藏了外部标识符。
22 (adv)	8.7 (req)	如果对象的访问只是在单一的函数中，那么对象应该声明在块范围内声明。
23 (adv)	8.10 (req)	在文件范围内声明和定义的所有对象或函数应该具有内部链接，除非是在需要外部链接的情况下。
24 (req)	8.11 (req)	static存储类标识符具有内部链接的对象和函数的定义和声明。
25 (req)	8.9 (req)	具有外部链接的标识符应该具有准确的外部定义。
26 (req)	8.4 (req)	如果对象或函数被声明了多次，那么它们的类型应该是兼容的。
27 (adv)	8.8 (req)	外部对象或函数应该声明在唯一的文件中。
28 (adv)		已废除
29 (req)	5.4 (req)	标签（tag）名称必须是唯一的标识符。
30 (req)	9.1 (req)	所有自动变量在使用前都应被赋值。
31 (req)	9.2 (req)	应该使用大括号以指示和匹配数组和结构的非零初始化构造。
32 (req)	9.3 (req)	在枚举列表中，"="不能显式用于除首元素之外的元素上，除非所有的元素都是显式初始化的。
33 (req)	12.4 (req)	逻辑运算符 && 或 || 的右手操作数不能包含副作用。
34 (req)	12.5 (req)	逻辑 && 或 || 的操作数应该是primary-expressions。
35 (req)	13.1 (req)	赋值运算符不能使用在产生布尔值的表达式上。

MISRA-C :1998	MISRA-C :2004	MISRA-C:2004 规则
36 (adv)	12.6 (adv)	逻辑运算符（&&、|| 和 !）的操作数应该是有效的布尔数。有效布尔类型的表达式不能用做非逻辑运算符（&&、|| 和 !）的操作数
37 (req)	10.5 (req)	如果位运算符 ~ 和 << 应用在基本类型为unsigned char或unsigned short的操作数，结果应该立即强制转换为操作数的基本类型。
37 (req)	12.7 (req)	位运算符不能用于基本类型（underlying type）是有符号的操作数上。
38 (req)	12.8 (req)	移位运算符的右手操作数应该位于零和某数之间，这个数要小于左手操作数的基本类型的位宽。
39 (req)	12.9 (req)	一元减运算符不能用在基本类型无符号的表达式上。
40 (adv)	12.3 (req)	不能在具有副作用的表达式中使用sizeof运算符。
41 (adv)	3.3 (adv)	应该确定、文档化和重视所选编译器中整数除法的实现。
42 (req)	12.10 (req)	不要使用逗号运算符。
43 (req)	10.1 (req)	下列条件成立时，整型表达式的值不应隐式转换为不同的基本类型： a) 转换不是带符号的向更宽整数类型的转换，或者 b) 表达式是复杂表达式，或者 c) 表达式不是常量而是函数参数，或者 d) 表达式不是常量而是返回的表达式。
44 (adv)		已废除
45 (req)	11.1 (req)	转换不能发生在函数指针和其他除了整型之外的任何类型指针之间。
45 (req)	11.2 (req)	对象指针和其他除整型之外的任何类型指针之间、对象指针和其他类型对象的指针之间、对象指针和void指针之间不能进行转换。
45 (req)	11.3 (adv)	不应在指针类型和整型之间进行强制转换
45 (req)	11.4 (adv)	不应在某类型对象指针和其他不同类型对象指针之间进行强制转换。
45 (req)	11.5 (req)	如果指针所指向的类型带有const或volatile限定符，那么移除限定符的强制转换是不允许的。
46 (req)	12.2 (req)	表达式的值在标准所允许的任何运算次序下都应该是相同的。
47 (adv)	12.1 (adv)	不要过分依赖C表达式中的运算符优先规则
48 (adv)	10.4 (req)	浮点类型复杂表达式的值只能强制转换到更窄的浮点类型。

MISRA-C :1998	MISRA-C :2004	MISRA-C:2004 规则
49 (adv)	13.1 (adv)	数的非零检测应该明确给出，除非操作数是有效的布尔类型。
50 (req)	13.2 (req)	浮点表达式不能做相等或不等的检测。
51 (adv)	12.11 (adv)	无符号整型常量表达式的计算不应产生折叠（wrap-around）。
52 (req)	14.1 (req)	不能有不可到达（unreachable）的代码。
53 (req)	14.2 (req)	所有非空语句（non-null statement）应该： a) 不管怎样执行都至少有一个副作用（side-effect），或者 b) 可以引起控制流的转移
54 (req)	14.3 (req)	在预处理之前，空语句只能出现在一行上；其后可以跟有注释，假设紧跟空语句的第一个字符是空格。
55 (adv)		已废除
56 (req)	14.4 (req)	不应使用goto语句。
57 (req)	14.5 (req)	不应使用continue语句。
58 (req)		已废除
59 (req)	14.8 (req)	组成switch、while、do...while或for结构体的语句应该是复合语句。
59 (req)	14.9 (req)	if（表达式）结构应该跟随有复合语句。else关键字应该跟随有复合语句或者另外的if语句。
60 (adv)	14.10 (req)	所有的if ... else if结构应该由else子句结束。
61 (req)	15.1 (req)	switch 标签只能用在当最紧密封闭（closely-enclosing）的复合语句是switch语句体的时候
61 (req)	15.2 (req)	无条件的break语句应该终止每个非空的switch子句。
62 (req)	15.3 (req)	switch语句的最后子句应该是default子句。
63 (adv)	15.4 (req)	switch表达式不应是有效的布尔值。
64 (req)	15.5 (req)	每个switch语句至少应有一个case子句。
65 (req)	13.4 (req)	for语句的控制表达式不能包含任何浮点类型的对象。
66 (adv)	13.5 (req)	for语句的三个表达式应该只关注循环控制。
67 (adv)	13.6 (req)	for循环中用于迭代计数的数值变量不应在循环体中修改。
68 (req)	8.6 (req)	函数应该声明为具有文件作用域。
69 (req)	16.1 (req)	函数定义不得带有可变数量的参数
70 (req)	16.2 (req)	函数不能调用自身，不管是直接还是间接的。
71 (req)	8.1 (req)	函数应当具有原型声明，且原型在函数的定义

MISRA-C :1998	MISRA-C :2004	MISRA-C:2004 规则
		和调用范围内都是可见的。
72 (req)	8.3 (req)	函数的每个参数类型在声明和定义中必须是等同的，函数的返回类型也该是等同的。
73 (req)	16.3 (req)	在函数的原型声明中应该为所有参数给出标识符
74 (req)	16.4 (req)	函数的声明和定义中使用的标识符应该一致
75 (req)	8.2 (req)	不论何时声明或定义了一个对象或函数，它的类型都应显式声明。
76 (req)	16.5 (req)	不带参数的函数应当声明为具有void类型的参数
77 (req)	10.2 (req)	下列条件成立时，浮点类型表达式的值不应隐式转换为不同的类型： a) 转换不是向更宽浮点类型的转换，或者 b) 表达式是复杂表达式，或者 c) 表达式是函数参数，或者 d) 表达式是返回表达式。
78 (req)	16.6 (req)	传递给一个函数的参数应该与声明的参数匹配。
79 (req)		已废除
80 (req)		已废除
81 (adv)	16.7 (adv)	函数原型中的指针参数如果不是用于修改所指向的对象，就应该声明为指向const的指针。
82 (adv)	14.7 (req)	一个函数在其结尾应该有单一的退出点。
83 (req)	16.8 (req)	带有non-void返回类型的函数其所有退出路径都应具有显式的带表达式的return语句。
84 (req)	Rescinded	已废除
85 (adv)	16.9 (req)	函数标识符的使用只能或者加前缀&，或者使用括起来的参数列表，列表可以为空。
86 (adv)	16.10 (req)	如果函数返回了错误信息，那么错误信息应该进行测试。
87 (req)	8.5 (req)	头文件中不应有对象或函数的定义。
87 (req)	19.1 (adv)	文件中的#include语句之前只能是其他预处理指令或注释。
88 (req)	19.2 (adv)	#include 指令中的头文件名字里不能出现非标准字符。
89 (req)	19.3 (req)	#include 预处理指令应该跟随 <filename> 或 "filename"序列。
90 (req)	19.4 (req)	C的宏只能扩展为用大括号括起来的初始化、常量、小括号括起来的表达式、类型限定符、存储类标识符或do-while-zero结构。

MISRA-C :1998	MISRA-C :2004	MISRA-C:2004 规则
91 (req)	19.5 (req)	宏不能在块中进行 #define和 #undef。
92 (adv)	19.6 (req)	不要使用#undef。
93 (adv)	19.7 (adv)	函数的使用优先选择函数宏。
94 (req)	19.8 (req)	函数宏的调用不能缺少参数
95 (req)	19.9 (req)	传递给函数宏的参数不能包含看似预处理指令的标记。
96 (req)	19.10 (req)	在定义函数宏时，每个参数实例都应该以小括号括起来，除非它们做为#或##的操作数。
97 (adv)	19.11 (req)	预处理指令中所有宏标识符在使用前都应先定义，除了#ifdef和#ifndef指令及defined()操作符。
98 (req)	19.12 (req)	在单一的宏定义中最多可以出现一次 # 或 ## 预处理器操作符。
98 (req)	19.13 (adv)	不要使用# 或 ## 预处理器操作符。
99 (req)	3.4 (req)	所有#pragma指令的使用应该文档化并给出良好解释。
100 (req)	19.14 (req)	defined预处理操作符只能使用两种标准形式之一。
101 (adv)	17.1 (req)	指针的数学运算只能用在指向数组或数组元素的指针上。
101 (adv)	17.2 (req)	指针减法只能用在指向同一数组中元素的指针上。
101 (adv)	17.4 (req)	数组的索引应当是指针数学运算的唯一可允许的方式
102 (adv)	17.5 (adv)	对象声明所包含的间接指针不得多于2级
103 (req)	17.3 (req)	>、>=、<、<= 不应用在指针类型上，除非指针指向同一数组。
104 (req)		已废除
105 (req)		已废除
106 (req)	17.6 (req)	自动存储对象的地址不应赋值给其他的在第一个对象已经停止存在后仍然保持的对象。
107 (req)		已废除
108 (req)	18.1 (req)	所有结构与联合的类型应该在转换单元（translation unit）的结尾是完善的。
109 (req)	18.2 (req)	对象不能赋值给重叠（overlapping）对象。
109 (req)	18.3 (req)	不能为了不相关的目的重用一块内存区域。
110 (req)	18.4 (req)	不要使用联合。
111 (req)	6.4 (req)	位域只能被定义为unsigned int或singed int类型。
112 (req)	6.5 (req)	unsigned int类型的位域至少应该为2 bits长度。
113 (req)		已废除

MISRA-C :1998	MISRA-C :2004	MISRA-C:2004 规则
114 (req)	20.1 (req)	标准库中保留的标识符、宏和函数不能被定义、重定义或取消定义。
115 (req)	20.2 (req)	不能重用标准库中宏、对象和函数的名字。
116 (req)	3.6 (req)	产品代码中使用的所有库都要适应本文档给出的要求，并且要经过适当的验证。
117 (req)	20.3 (req)	传递给库函数的值必须检查其有效性。
118 (req)	20.4 (req)	不能使用动态堆的内存分配
119 (req)	20.5 (req)	不要使用错误指示errno。
120 (req)	20.6 (req)	不应使用库<stddef.h>中的宏offsetof。
121 (req)		已废除
122 (req)	20.7 (req)	不应使用setjmp宏和longjmp函数
123 (req)	20.8 (req)	不应使用信号处理工具<signal.h>
124 (req)	20.9 (req)	在产品代码中不应使用输入/输出库<stdio.h>。
125 (req)	20.10 (req)	不应使用库<stdlib.h>中的函数atof、atoi和atol。
126 (req)	20.11 (req)	不应使用库<stdlib.h>中的函数abort、exit、getenv和system。
127 (req)	20.12 (req)	不应使用库<time.h>中的时间处理函数。

Appendix C:MISRA-C:1998 – 已废除的规则

Rule 8 (rescinded):	Multibyte characters and wide string literals shall not be used.
Rule 18 (rescinded):	Integer constants should be suffixed to reflect their type where an appropriate suffix is available.
Rule 20 (rescinded):	All object and function identifiers shall be declared before use.
Rule 28 (rescinded):	The register storage class specifier should not be used.
Rule 44 (rescinded):	Redundant explicit casts should not be used.
Rule 55 (rescinded):	Labels should not be used, except in switch statements.
Rule 58 (rescinded):	The break statement shall not be used (except to terminate the cases of a switch statement).
Rule 79 (rescinded):	The values returned by void functions shall not be used.
Rule 80 (rescinded):	Void expressions shall not be passed as function parameters.
Rule 84 (rescinded):	For functions with void return type, return statements shall not have an expression.
Rule 104 (rescinded):	Non-constant pointers to functions shall not be used.
Rule 105 (rescinded):	All the functions pointed to by a single pointer to function shall be identical in the number and type of parameters and the return type.
Rule 107 (rescinded):	The null pointer shall not be de-referenced.
Rule 113 (rescinded):	All the members of a structure (or union) shall be named and shall only be accessed via their name.
Rule 121 (rescinded):	<locale.h> and the setlocale function shall not be used.

Appendix D：ISO 标准交互参考

本附录给出文档中提供的规则与 ISO 9899[2]的对应关系。

D1. MISRA-C :2004 规则序号与 ISO 9899 的对应关系

Rule	ISO ref	Rule	ISO ref	Rule	ISO ref
1.4	6.1.2	**8.2**	6.5.4	**12.1**	6.3
1.5	6.1.2.5	**83**	6.5.4.3,6.7.1	**12.2**	5.1.2.3, 6.3, 6.6
2.2	6.1.9	**8.4**	6.1.2.6,6.5	**123**	5.1.2.3,6.3.3.4
2.3	6.1.9	**8.5**	6.8.2	**12.4**	5.1.2.3,6.3.13,
2.4	6.1.9	**8.6**	6.1.2.1, 6.5.4.3		6.3.14
33	6.3.5	**8.7**	6.1.2.1,6.5	**12.5**	6.3.1,6.3.13,
3.4	6.8.6	**8.8**	6.7		6.3.14
3.5	6.5.2.1	**8.9**	6.7	**12.6**	6.3.3.3,6.3.13,
4.1	52.2	**8.10**	6.1.2.1,6.1.2.2,		6.3.14
4.2	5.2.1.1		6.5.4	**12.7**	6.3.3.3, 6.3.7,
5.1	6.1.2	**8.11**	6.1.2.2,6.5.1,		6.3.10,63.11
5.2	6.1.2,1		6.5.4		6.3.12
5.3	6.5.6	**9.1**	6.5.7	**12.8**	6.3.7
5.4	6.5.2.3	**9.2**	6.5.7	**12.9**	6..3.3..3
5.5	6.1.2.2	**9.3**	6.5.2.2	**12.10**	6.3.17
5,6	6.1.2.3	**10.1**	6.2	**12.11**	6.4,'6.8.1
5.7	6.1.2	**10.2**	6.2	**12.12**	5.2.4.2.2, 6.1.2.5
6.1	6.1.2.5	**10.3**	6.2, 6.3.4	**12.13**	6.3.2.4,6.3.3.1
6.2	6.2.1.1		6.2, 6.3.4	**13.1**	6.3.16
63	6.1.2.5,6.5.2,	**10.5**	6.3.33, 6.3.7	**13.2**	6.6.4.1,6.6.5
	6.5.6	**10.6**	6.1.3.2	**133**	5.2.4.2.2, 6.3.9
6.4	6.5.2.1	**11.1**	6.3.4	**13.4**	6.6.5.3
6.5	6.5.2.1	**11.2**	6.3.4	**134**	6.6.5.3
		11.3	6.3.4	**13.6**	6.6.5.3
7.1	6.1.3.2	**11.4**	6.3.4	**14 2**	5.1,23, 6.6.3
8.1	6.3.2.2, 6.5.4.3	**11.5**	6.53	**143**	6.6.3
				14.4	6.6.6.1
14.5	6.6.6.2	**17.2**	6.3.6, 6.3.16.2	**19.12**	6.8.3.2, 6.8.3.3
14.6	6.6.6.3	**173**	6.3.8	**19.13**	6.8.3.2, 6.8.3.3
14.7	6.6.6.4	**17.4**	6.3.2.1,6.3.6	**19.14**	6.8.1
14.8	6.6.5	**17.5**	6.33.2, 6.5.4.1	**19.15**	6.8
14.9	6.6.4.1	**17.6**	6.1.2.4,6.33.2	**19.16**	6.8
14.10	6.6.4.1	**1.8.1**	6.1.2.5,6.5,	**20.1**	6.87,1.3.7.13
15.1	6.6.4.2		6.5.2.1	**20.2**	7.1
15.2	6.6.4.2,	**1.8.2**	6.3.16.1	**20.3**	7.17.3,7.5.1
15.3	6.6.4.2	**1.8.4**	6.1.2.5,6.3.2.3,		7.5.6.4
15.4	6.6.4.2		6.5.2.1	**20.4**	7.1
15.5	6.6.4.2	**19.1**	6.8.2	**20.5**	7.17.5.1
16.1	6.7.1,	**19.2**	6.1.7	**20.6**	7.1
16.2	6.3.2.2	**19.3**	6.8.2	**20.7**	7.6
16.3	6.5.4.3	**19.4**	6.8.3	**20.8**	7.7

16.4	6.5.4.3	**19.5**	6.8.3,6.8.3.5	**20.9**	7.9
16.5	6.5.4.3,	**19.6**	6.8.3.5	**20.10**	7.10.1
16.6	6.3.2.2	**19.7**	6.8.3	**20.11**	7.10.4
16.7	6.5.4.1,	**19.8**	6.8.3	**20.12**	7.12
16.8	6.6.6.4	**19.9**	6.8.3	**21.1**	6.3.3.3.2,
16.9	6.3.2.2,	**19.10**	6.8.3		7.1
17.1	6.3.6,	**19.11**	6.8	**21.2**	7.1,

D1.ISO 9899 参考与 MISRA-C :2004 规则序号的对应关系

ISO ref	Rule	ISO ref	Rule	ISO ref	Rule
5.1.23	12.2, 123, 12.4, 14.2	6.3.6	17.1, 17.2, 17.4	6.6.5.3	13.4, 13.5,
		6.3.7	10.5, 12.7, 12.8	6.6.6.1	14.4
5.2.2	4.1	6.3:8	17.3.	6.6.6.2	14.5
5.2.1.1	4.2	6.3.9	13.3	6.6.6.3	14.6, 15.2
5.2.4.2.2	12.12, 13.3	6.3.10	12.7	6.6.6.4	14.7, 16.8
6.1.2	1.4,5.1,5.7	6.3.11	12.7	6.7	8.8, 8.9
6.1.2.1	5.2,8.6,8.7,8.10	6.3.12	12.7	6.7.1	8.3, 16.1, 16
6.U.2	5.5,8.10,8.11	6.3.13	12.4,12.5, 12.6		16.5
6.1.2.3	5.6	6.3.14	12.4, 12.5, 12.6	6.8	19.11, 19.15
6.1.2.4	17.6	6.3.16	13.1	6.8.1	12.11, 19.14
6.1.2.5	1.5,6.2,6.3, 12.12, 18.1, 18.4	6.3.16.1	18.2	6.8.2	8.5, 19.1, 19
		6.3.16.2	17.1, 17.2	6.8.3	19.4, 19.5,
6.1.2.6	8.4	6.3.17	12.10		19.7-10
6.1.3.2	7.1, 10.6	6.4	12.11	6.83.2	19.12,19.13
6.1.7	19.2	6.5	8.4, 8.7, 18.1	6.83.3	19.12, 1-9.13
6.1.9	2.2, 2.3, 2.4	6.5.1	8.11-	6.8.3.5	19.5,19.6
6.2	10.1, 10.2, 10.3, 10.4	6.5.2	6.3	6.8.6	3.4
		6.5.2.1	3.5, 6.4, 6.5, 18.1, 18.4	6.8.8	20.1
6.2.1.1	6.1			7.1.3	20.1, 20,2
6.3	111,12.2,21.1	6.5:2.2	9.3 T"	7.1.6	20.6
6.3.1	12.5	6.5.2.3	5.4	7.1.7	20.3
6.3.2.1	17.4	6.5.3	11.5	73	20.3
6.3.2.2	8.1,16.2, 16.6. 16.9	6.5.4	8.2,8.10,8.11	7.5.1	20.3, 20.5
		6.5.4.1	16:7, 17.5	7.5.6.4	20.3
6.3.2.3	18.4	6.5.4.3	8.1, 8.2,8.3; 8.6,	7.6	20.7
6.3.2.4	12.13		16.3, 16.4, 16.5,	7.7	20.8
6.3.3.1	12.13		16.7	7.8	16.1
6.33.2	16.9, 17.5, 17.6, 21.1	6.5.6	5.3, 6.3	7.9	20.9
		6.5.7	9.1,9.2-..	7.10.1	20.10
6.3.3.3	10.5, 12.6, 12.7, 12.9	6.6	12.2	7.10.3	20.4
		6.6.3	14.2,14.3	7.10.6	21.1
6.3.3.4	12.3	6.6.4.1	13.2, 14.9, 14.10	7.12	20.12
6.3.4	10.3,10.4, 11.1, 11.2, 11.3, 11.4	6.6.4.2	15.1 - 15.5	7.13	20.1
		6.6.5	13.2, 14.8		
6.3.5	3.3				

Appendix E： 术语表

基本类型（Underlying type）

见 6.10.4 节“基本类型”

规则范围

MISRA 规则适用于工程项目中的所有源文件。即：

- 由项目小组编写的文件
- 所有由其他小组和副代理商提供的文件
- 编译器库的头文件，但不是库文件，后者只做为目标代码
- 所有第三方的库资源，特别是头文件

要认识到，在获得第三方资源符合规则标准时可能会有些问题，但许多工具和库的供应商为他们的代码采用 MISRA-C 做为标准。

布尔表达式

严格说来，C 当中没有布尔类型，但返回数值类型的表达式和返回布尔类型的表达式存在着概念上的差异。一个表达式表示了布尔值，或者是因为它出现在需要布尔值的地方，或者是因为它使用了产生布尔值的运算符。

在下面的上下文环境中希望布尔值：

- if 语句的控制表达式
- 迭代语句的控制表达式
- 条件运算符 ?的第一个操作数

每种环境需要一个“有效布尔”表达式，或者是“Boolean-by-construct”，或者是“Boolean-by-construct”，定义如下：

Boolean-by-contruct 值由以下运算符产生：

- 等值运算符（== 和 !=）
- 逻辑运算符（ !，&& 和 ||）
- 关系运算符（<，>，<= 和 >=）

Boolean-by-enforcement 值可以使用工具实现一个特定的强制类型来产生。此时布尔类型与特定的 typedef 相关联，可以用于任何布尔对象。这可以带来许多好处，尤其是当具有检查工具的支持时，而且，在某些特殊情况下它能帮助避免逻辑运算和整数运算之间的混淆。

短整型（smaller integer type）

术语“短整型”用来描述那些服从整数提升（interger promotion）的整数类型。受整数提升影响的类型是 char，short，bit-field 和 enum。